

Computer Science Department

TECHNICAL REPORT

Automated Inversion of a Unification
Parser into a Unification Generator

Tomek Strzalkowski

Technical Report 465

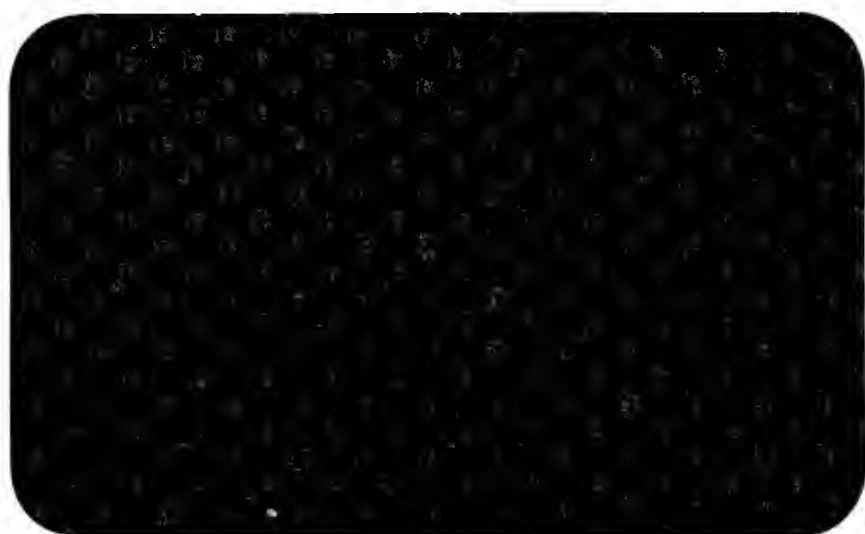
September 1989

NEW YORK UNIVERSITY



Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

NYU COMPSCI TR-465
Strzalkowski, Tomek
Automated inversion of a
unification parser into a
unification... c.2



**Automated Inversion of a Unification
Parser into a Unification Generator**

Tomek Strzalkowski

Technical Report 465

September 1989

Automated inversion of a unification parser into a unification generator

Tomek Strzalkowski
Courant Institute of Mathematical Sciences
New York University
715 Broadway, rm. 704
New York, NY 10003

Abstract - This paper describes an algorithm for an automated conversion of a unification parser for natural language into an efficient unification generator for the same language. The scope of applicability of the algorithm is discussed and possible avenues of extension are suggested. The algorithm is tested on an actual grammar derived from a larger string grammar for English.

1. Motivation

The purpose of this research is to explore possibilities of an automated derivation of an efficient unification-based generator for a natural language (English, Japanese) from a unification-based parser for the same language. Although the idea of reversible Prolog programs is not a new one, there has always been the problem of efficiency. This paper describes some preliminary results obtained from the experiments with reversing a Prolog parser for a substantial subset of English into an efficient generator. Similar experiments with Japanese parser are under way. The starting point of the experiment is a string grammar for English developed by the NYU's Linguistic String Project (Sager 1981) and subsequently used as a basis of the string parser in the Proteus Project (Grishman, et al.). The string parser, which produces a regularized operator-argument parse, is first converted into an equivalent unification parser written in Prolog. This involves translating the BNF part of the string grammar (CF syntax + regularization rules) as well as the restriction component (specifying context-sensitive dependencies). The conversion into Prolog is performed according to certain principles which are aimed at assuring a "reversibility" of so-obtained parser. The Prolog parser can be subsequently "compiled" into an efficient Prolog generator working backwards: from regularized parse forms to English sentences. To obtain a Prolog parser (or a Prolog program, in general) working in the reverse, requires (barring the presence of non-reversible operations, such as arithmetic operations) some manipulation of the clauses, especially the ordering of the literals on their right-hand side. On the 1988 CMU MT conference, Dymetman and Isabelle proposed a semi-automatic process in which the non-terminals in the right-hand side of the productions in the source grammar are explicitly marked (manually) for the order in which they should be expanded in generation. The annotated BNF grammar is then used to derive both a Prolog parser and a Prolog generator. However, the semantic component of the grammar is not fully integrated into the inversion process. The effect is that, at least in certain cases, the generator must rely on dynamic goal ordering forcing it to "freeze" some goals until certain variables get instantiated. The goal-freezing mechanism, however, adds a substantial overhead during execution. The approach taken in the present experiment is to convert the Prolog parser in which both the semantic component as well as the context-sensitive restrictions have been accommodated into a single executable program. The goals of this research can be stated as follows:

- (1) Describe the conditions under which the unification-based parser is reversible to form an efficient generator without the need to write a separate program. This includes the problem of reversibility of the underlying grammar formalism on which the parser is based.
- (2) Specify transformations necessary to automatically convert a parser satisfying the conditions set up in (1) into a generator that would be at least as efficient the parser.¹

2. Converting a string grammar into Prolog

A string grammar used in the Proteus Project consists of (1) context-free BNF component, (2) "semantic" rules which construct the regularized parse (semantic rules accompany the CF rules), and (3) the restriction component specifying context-sensitive and other constraints imposed upon the CF component, such as number agreement between phrases and verb subcategorization (types of object strings verbs can accept), and so forth. The CF

¹ In the present experiment an average processing time for a 15-word sentence ranged between 0.10 and 0.70 seconds per parse (for parsing), and between less than 0.01 and 0.25 seconds per sentence (for generation).

component is transformed into Prolog using a standard DCG method of furnishing the non-terminals with variables binding the "input" and "output" strings. Thus, the BNF production $N ::= N1 N2 N3$ is translated into the following definite clause:

```
n(X1,X4) :- n1(X1,X2),n2(X2,X3),n3(X3,X4).
```

The "semantic" rules accompanying the CF productions come in two types: local and non-local. The local rules combine the translations of some of the non-terminals on the right-hand side of a production into the translation of the non-terminal on the left. These can be incorporated into the Prolog parser by adding additional literals at the end of the clause, for example,

```
n(X1,X4,P) :- n1(X1,X2,P1),n2(X2,X3,P2),n3(X3,X4,P3),combine(P1,P2,P3,P).
```

where the variables P, P1, P2 and P3 will be bound to "translations" of N, N1, N2 and N3, respectively. Non-local "semantic" rules create translations of left-hand non-terminals out of elements that may not be available on the right-hand side of the production, but instead are passed over from other productions. For example, in

```
n(X1,X4,P) :- n1(X1,X2,P1),n2(X2,X3,P2),n3(X3,X4,P1,P2,P).
```

the "semantics" of n3 is not formed until the actual expansion of n3 is known. Because the "semantics" of n1 and n2 are to be included in the final translation for n, these (P1 and P2) are passed as arguments to n3, and so is P. Finally, the restriction component is incorporated into the parser. At this time only a limited amount of restrictions have been added, including some of the restrictions on the types of object strings that certain verbs can take, the number agreement and some constraints governing the placement of sentence adjuncts.

2.1. Examples of translating a grammar with semantic rules

A production in the string grammar consists of the context-free part and the "semantic" part specifying how the "translations" of the components are to be combined to create the translation of the head non-terminal on the right-hand side.

$N ::= N1 N2 N3$:<form translation of N out of those for N1, N2 and N3>

There are two basic ways of specifying the translation of N: local and non-local. The local translation, or local semantics, composes the translation of N out of translations of N1, N2 and N3 in such a way that the latter three become arguments of some constant function f. Productions with local semantics may take the following general form:

$N ::= N1 N2 N3$:f(N1',N2',N3')

where Ni' denotes the "semantics" (i.e., translation) of Ni. In this case, the translation into Prolog is simple:

```
n(X1,X4,f(P1,P2,P3)) :- n1(X1,X2,P1),n2(X2,X3,P2),n3(X3,X4,P3).
```

As an example, consider (a somewhat simplified) production recognizing the left adjunct phrase to a noun:

```
<ln> ::= <tpos> <qpos> <apos> <npos>
      :(<tpos> <qpos> <apos> <npos>).
```

This rule is converted into the following Prolog clause:

```
ln(L1,L5,[P1,P2,P3,P4]) :-
    tpos(L1,L2,P1),
    qpos(L2,L3,P2),
    apos(L3,L4,P3),
    npos(L4,L5,P4).
```

Sometimes, when the function f is not as simple as the list-building primitive, we may want to use a non-local translation (see below). For example, if lists translating adjunct components need to be spliced in rather than merely juxtaposed, we may require the following Prolog clause instead:

```
ln(L1,L5,P) :-
    tpos(L1,L2,P1),
    qpos(L2,L3,P2),
    apos(L3,L4,P3),
    npos(L4,L5,P4),
    #([P1,P2,P3,P4],P).
```

Productions with non-local semantics occur when the translation of the head non-terminal is constructed out of elements which are unknown or unavailable to the production, or require special handling by dedicated routines (such as splice-in # above). Such productions give rise to the so-called "chain rules"² where the translation of the head non-terminal is not constructed immediately but instead it is passed, along with the translations of some of the right-hand side elements, to some other production or a dedicated routine for the final assembly. Except for the case illustrated above, rules with non-local "semantics" usually involve at least two productions, which may take the following general form:

$$\begin{array}{ll} N ::= N1 N2 N3 & : \lambda P P(N1', N2') (N3') \\ N3 ::= N4 & : \lambda P \lambda Q Q(P, N4') \end{array}$$

where Ni' denotes "semantics" (or translation) of Ni , and P and Q are variables. The corresponding Prolog clauses are as follows:

```
n(X1,X4,P) :- n1(X1,X2,P1),n2(X2,X3,P2),n3(X3,X4,P1,P2,P).
n3(X1,X2,P1,P2,P) :- n4(X1,X2,P3),combine([P1,P2,P3],P).
```

or

```
n(X1,X4,P) :- n1(X1,X2,P1),n2(X2,X3,P2),n3(X3,X4,P1,P2,P).
n3(X1,X2,P1,P2,f(P1,P2,P3)) :- n4(X1,X2,P3).
```

For example, the following two productions from the string grammar:

$$\begin{array}{ll} \langle \text{assertion} \rangle & ::= \langle \text{subject} \rangle \langle \text{verb} \rangle \langle \text{object} \rangle \\ & : (\langle \text{object} \rangle \langle \text{subject} \rangle \langle \text{verb} \rangle). \\ \langle \text{object} \rangle & ::= \langle \text{nstgo} \rangle \\ & : (\text{lambda (subj vb)} \\ & \quad (\text{!vb (subject subj) (object } \langle \text{nstgo} \rangle))) \end{array}$$

are translated into two clauses in Prolog:

```
assertion(A1,A4,P) :-
    subject(A1,A2,P1),
    verb(A2,A3,P2),
    object(A3,A4,P1,P2,P).
object(O1,O2,P1,P2,[P2,[subject,P1],[object,P3]]) :- nstgo(O1,O2,P3).
```

The chain rule for assertion passes an unbound copy of the translation variable P to the object clause, along with bound translations of subject and verb, $P1$ and $P2$. Object clause, when called, accommodates the latter two and its own translation $P3$ into the final translation of assertion which subsequently binds variable P .

The reader may note that the chain rule featured in this example could be reduced to a non-chain rule with local semantics by eliminating non-terminal *object* and rewriting the clause defining *assertion* as follows:

```
assertion(A1,A4,[P2,[subject,P1],[object,P3]]) :-
    subject(A1,A2,P1),
    verb(A2,A3,P2),
    nstgo(A3,A4,P3).
```

We call this move *path compression* because it shortens some paths in the parse tree by eliminating non-terminals from the grammar. Path compression can be used to simplify the parser code during the code normalization process described in section 4. One obvious reason for doing so is the fact that non-chain, local-semantics rules frequently need no further processing³ and can be used directly for generation. Even so, path compression must be used carefully, or it may have an adverse effect on the program performance. For one thing, path compression will almost certainly decrease the efficiency of the parser, since it would replace common parts of certain paths by a number of disjoint paths, thus increasing the potential backup ratio. Consequently, it may decrease the efficiency of the generator if the same parse structure can activate several alternative routes for generation. This latter problem can be averted by merging these paths as long as they carry the same, unique parse structures. The problem of path

² We borrow this term from Shieber et al. (1989).

³ Inversion may still be required for efficiency reasons, if arguments to the literals remain in certain mutual dependencies.

compression and merging is further discussed in section 4.3.

2.2. Translating context-sensitive restrictions in the string grammar

An important part of a string grammar are context-sensitive, and other, restrictions imposed upon context-free productions of the grammar in order to narrow their applicability in certain cases. Among these restriction are the number agreement restriction between subject and the first verb of sentences, various restrictions on the placement of sentence adjuncts, restrictions on the types of object strings a verb may accept, and so forth. At present, no general method of translating these restrictions into Prolog code is proposed, and many of them have been accommodated on the one-by-one basis. There is work currently in progress to make this translation fully automatic too.

3. Reversing a Prolog parser for generation

In principle, a Prolog parser can work in reverse, that is, given a regularized operator-argument form it can generate a sentence, or sentences, that would have been parsed into this form. In practice, however, a bidirectional parser/generator is not a workable idea, and the parser program needs to be "tuned" for generation. A number of specific translation rules have been developed that transform the source of the Prolog parser into a generator. There are generally two kinds of rules: (1) these that actually reverse the parser by changing the order in which literals in a clause are expanded, and (2) those that prepare the parser for inversion compiling it into a "normal" form by replacing groups of clauses by new clauses in order to (a) assure invertibility of the parser and (b) enhance the efficiency of the inverted program. We discuss the inversion process first, ignoring for a moment the problem of non-invertible clauses. Later, we point out some "non-normal" productions that may occur in the grammar, and subsequently in the parser, and which cannot be coped with adequately by the inversion algorithm. We suggest ways how these can be normalized by a pre-processing compilation stage.

The way in which the inversion process affects the parser's clauses (which correspond to productions in the grammar) depends on how the "semantics" of the head-literal non-terminal of this clause have been defined. For the clauses with a local "semantics", no changes need to be done for their use in the generator. A clause with a non-local "semantics", on the other hand, needs to be replaced by a new clause with the same head literal, but with its right-hand side literals appropriately rearranged. For example, the clause

$$n(X1,X4,P) :- n1(X1,X2,P1),n2(X2,X3,P2),n3(X3,X4,P1,P2,P).$$

needs to be replaced by the new one in which the literal $n3(X3,X4,P1,P2,P)$ appears at the front of the right-hand side, in other words, it is *fronted*. The reason for this is that during generation the binding of P is known before the bindings to any other variables in this clause. Because P is constructed out of (among others) $P1$ and $P2$, these will have known bindings when $n3$ returns. Subsequently, no other changes are required. In general case, however, the process of literal fronting may require recursive application to the remaining literals.

3.1. Some examples of literal reordering

The grammar rules with their "semantics" defined locally, frequently do not need to be modified at all for the use in a generator. A parser clause of the form given below, for example, needs no tuning for the use in a generator.

$$n(X1,X3,[P1|P2]) :- n1(X1,X2,P1),n2(X2,X3,P2).$$

On the other hand, if any of the literals on the rhs of a clause contains the variable that is expected to be bound to the "translation" of the non-terminal on the lhs, then this literal must be fronted on rhs, i.e., it must be fired first during generation. Thus having a clause of the form:

$$n(X1,X3,P) :- n1(X1,X2,P1),n2(X2,X3,P1,P).$$

we have to replace it by

$$n(X1,X3,P) :- n2(X2,X3,P1,P),n1(X1,X2,P1).$$

Similarly,

$$n(X1,X4,P) :- n1(X1,X2,P1),n2(X2,X3,P2),n3(X3,X4,P3),combine([P1,P2,P3],P).$$

is replaced by

$$n(X1,X4,P) :- combine([P1,P2,P3],P),n1(X1,X2,P1),n2(X2,X3,P2),n3(X3,X4,P3).$$

The reader may note that fronting of *combine* makes sense only under the assumption that this predicate can be run backwards itself, that is, with only the argument P bound. If this is not the case, though, we have to modify the

clause(s) defining *combine* by manipulating its (their) rhs's, so that it starts behave more like *decompose*.

3.2. A general algorithm for literal reordering

The algorithm presented below describes a general method of rearranging the literals in the right-hand side (rhs) of a Prolog clause. As the result of applying this algorithm to a Prolog program, one obtains a fairly efficient "inverted" program. In many cases the obtained inverted program is optimal, that is, as efficient as as it can be given the set of available transformations that can be applied to the original: clause replacement, clause reordering and literal reordering. The algorithm does not deal with irreversible built-in Prolog primitives such as arithmetics.

3.2.1. Essential arguments

Some arguments of every literal are essential in the sense that the literal cannot be executed successfully unless all of them are bound, at least partially, at the time of execution. For example, the literal `concat(X,Y,Z)` that concatenates lists X and Y into list Z can be executed only if either X or Z is bound. An argument is considered *fully bound* if it is a constant or it is bound by a constant; an argument is *partially bound* if it is, or is bound by, a functional expression (not a variable) in which at least one variable is unbound. We may also define the degree to which an argument is bound (see the following section). If `concat` is used to concatenate lists then X must be bound; if it is used to decompose lists then Z must be bound. In general, a literal may have several alternative (possibly overlapping) sets of essential arguments. If all arguments in any one of such sets of essential arguments are bound, then the literal can be executed. Any set of essential arguments which have the above property is called *essential*. We shall call the set MSEA of essential arguments a *minimal set of essential arguments* if it is essential, and no proper subset of MSEA is essential. If any of MSEA's of a given literal is an empty set, then this literal needs no essential arguments, which means it can be executed at any time, even if none of its arguments is bound.⁴ Out of many different MSEA's a literal may have two subsets will be of special interest to us: these which define the direction in which the literal is used: forward or backward. `Concat(X,Y,Z)`, for example, has two MSEA's: $MSEA1 = \{X\}$ and $MSEA2 = \{Z\}$. When the variable in MSEA1 is bound, then `concat` is used forward, i.e., for concatenation. When the variable in MSEA2 is bound, on the other hand, then `concat` is used to decompose. In this way, the concept of invertibility appears well defined. Literals that have less than two MSEA's are not reversible but then there may be no need to reverse them. However, their presence in a Prolog program affects the reversibility of this entire program in dramatically different ways. 1-MSEA literals must always have their only MSEA variables bound before they can be executed. If the only MSEA is a non-empty set, then the literal is not reversible and must be used in one direction only. However, if a 1-MSEA literal has an empty set of MSEA's then it can be executed at any time whatever. Arithmetic operations are examples of 1-MSEA literals with a non-empty MSEA. In the domain of parsing, lexicon look-up primitives that check properties of specific words are practically irreversible. Among 1-MSEA literals with an empty MSEA probably the most common are those recognizing specific terminals in the input, such as `to`'s `to(X,Y)` or `that`'s `that(X,Y)` literals, which simply remove word "to" and "that", respectively, from the input string. 0-MSEA literals, that is, those that have no MSEA at all, can never be executed meaningfully, and thus must be considered ill-formed.

So defined, the notion of the set of essential arguments can be used to determine the invertibility of a Prolog program, but the concept is still too weak to be of any practical use. It seems that any literal with at least two MSEA's can be reversed, but, as we shall see, this requirement can be further relaxed. This can be accomplished by introducing a generalized notion of a MSEA. Before we move to do this, however, we need to define two auxiliary concepts of "in" and "out" arguments.

3.2.2. In and out arguments

Arguments in a literal can also be marked as being either "in" or "out" depending on whether or not they are bound at the time the literal is executed in a specific direction. Thus, in `concat([a,b],[c,d],Z)`, the first two arguments are "in", while the third is "out". The roles are reversed when `concat` is used for decomposition, as in `concat(X,Y,[a,b,c,d])`. In a more general case, the roles of "in" and "out" arguments do not exactly reverse when the direction of computation is reversed. It may be noted, however, that the role reversal is expected for 2-MSEA

⁴ An empty MSEA should not be confused with the situation where a literal has no MSEA, meaning it could never be evaluated successfully, even though all of its arguments were bound. In the former case, the set of MSEA's contains an empty MSEA as one of its elements; in the latter case, the set of MSEA itself is empty.

literals with disjoint MSEA's. As an example consider the literal $\text{subject}(A1,A2,\text{WHQ},\text{NUM},P)$ where $A1$ and $A2$ are input and output strings of words, WHQ indicates whether the subject phrase is a part of a clause within a wh-question, NUM is the number of the subject phrase, and P is the final translation. Out of these arguments only two are essential: $A1$ when parsing and P when generating. In parsing, the "in" arguments are: $A1$ and WHQ , the "out" arguments are $A2$, NUM and P ; in generating, the "in" arguments are P and WHQ , the "out" arguments are $A1$ and NUM . In generating, $A2$ is neither "in" nor "out". Thus, upon reversing the direction of computation, an "out" argument does not automatically become an "in" argument, nor does an "in" argument automatically become an "out" argument. Below is the general algorithm for computing "in" and "out" status of arguments in any given literal in a Prolog program.

An argument X of literal $\text{pred}(\dots X \dots)$ on the rhs of a clause is "in" if

- (A) it is a constant; or
- (B) it is a function and all its arguments are "in"; or
- (C) it is "in" or "out" in some previous literal on the rhs of the same clause, i.e., $I(Y) :- r(X,Y), \text{pred}(X)$; or
- (D) it is "in" in the head literal L on lhs of the same clause.

An argument X is "in" in the head literal $L = \text{pred}(\dots X \dots)$ of a clause if (A), or (B), or

- (E) L is the top-level literal and X is "in" in it (known apriori); or
- (F) the argument Y occupying X 's position is "in" in all occurrences of pred on the rhs of any clause with head predicate pred1 different than pred , and such that Y unifies with X .

A similar algorithm can be proposed for computing "out" arguments:

- (R) An argument X of literal $\text{pred}(\dots X \dots)$ on the rhs of a clause is "out" if it is "in" in $\text{pred}(\dots X \dots)$, or the argument Y occupying X 's position and unifiable with X in the lhs of all clauses with the head literal $\text{pred}(\dots Y \dots)$ is either "in" or "out".
- (L) An argument X of literal $\text{pred}(\dots X \dots)$ on the lhs of a clause is "out" if it is "in" in $\text{pred}(\dots X \dots)$, or it is "out" in literal $\text{pred1}(\dots X \dots)$ on the rhs of this clause, providing that $\text{pred1} \neq \text{pred}$ (again, we must take provisions to avoid infinite descend, c.f. (F) in "in" algorithm).

In addition, we may note that an argument which is a functional expression (containing variables and possibly further functional expressions) is (fully) "out" iff each of its own arguments is (fully) "out".

At first, the above steps may appear too weak to determine "out" status for arguments in recursive definitions. For instance, we do not know whether a recursion stops and how the arguments are bound until we actually trace the entire process and see that the recursion stops successfully. However, if it does not stop successfully, i.e., it either fails or loops forever, then the question of "out" status of arguments involved becomes irrelevant. On the other hand, if it does stop, our method will correctly predict the status of "out" arguments. As an example, consider the following recursive situation:

```

I(...) :- ...,pred(A,B),...
....
pred(fun(X),Y) :- pred(X,Y).
pred(const,const).

```

The algorithm presented above will predict that B is an "out" argument in $\text{pred}(A,B)$ on the rhs of the first clause (providing the two-clause definition below comprises of all the clauses with pred on lhs). Obviously, the only way the recursive rule is going to stop without failure is to fire the stop condition provided by $\text{pred}(\text{const},\text{const})$. In the case this does not happen, we are not interested in the status of A and B at all. In addition, we ignore all clauses that contain an explicit fail literal among their rhs literals.

In practice we do not need this elaborate general algorithm, and the problem of "in" and "out" arguments can be localized within a single clause. The simplified version of the above algorithm that will be used in the present experiment is as follows.

An argument X of literal $\text{pred}(\dots X \dots)$ on the rhs of a clause is "in" if

- (A) it is a constant; or
- (B) it is a function and all its arguments are "in"; or

(C) it is "in" or "out" in some previous literal on the rhs of the same clause, i.e., $l(Y) :- r(X,Y), \text{pred}(X)$; or

(D) it is "in" in the head literal L on lhs of the same clause (explicitly set).

Some comment in order, regarding the point (D) of the above algorithm. Although the "in" arguments of the head literal must be known before the right-hand side literals of this clause can be reordered, we do not specify how this information is to be obtained. The more general algorithm given earlier required full analysis of the static AND/OR space of the program to determine "in" arguments. For the simplified version of the algorithm we are left with two basic options. The first of these requires that we know apriori which arguments of the head literal of any clause are expected to be "in" during generation. The other option requires only that we know the "in" status of the arguments in the head of top-level clause. As the process of program inversion progresses from the top-level clause to the clauses invoked from it, the information about "in" arguments is passed along.

We also need a more practical method of computing "out" arguments. This is of a special significance because the ability to determine "out" arguments will be crucial for computing the generalized sets of essential arguments, as discussed in the following section. The centerpiece of the algorithm for determining "out" arguments in a Prolog clause is the recursive step of (R) above, with the stop condition provided by (L). Even though (L) may not look as a solid stop condition, we have been able to determine that there are actually three types of situations where the recursion stops with an argument X being marked as "out". The simplest case occurs when a variable X is bound to a constant expression as the result of the unification of the literal containing X with a lhs occurrence of the same literal containing a constant expression in X's position. Thus, in $\text{noun}(N1, N2, \text{Num}, P)$ the arguments Num and P are "out" because they are matched against constants in all lhs occurrences of noun, for example $\text{noun}([\text{john}|N], N, \text{sg}, \text{john})$. The second type of situation occurs when X is unified with an "in" argument. The general patterns is this: $\text{pred}(X, Y)$, with X "in" and Y's status unknown, is unified with $\text{pred}(f(X), X)$, and thus Y becomes "out". For example, in the $\text{noun}(N1, N2, \text{Num}, P)$ given above, N2 becomes "out" whenever N1 is "in". We may note here that the "out" status of an argument may depend on the "in" status of other arguments, and thus, ultimately, on the "direction" in which a program is run (Shoham and McDermott 1984). The final possibility is when a variable receives a partial binding, that is, it is bound by a functional term which may contain some unbound variables. In other words, $\text{pred}(X, Y)$ is unified with $\text{pred}(X, f(\text{const}, Z))$ or, in a still more general case, with $\text{pred}(X, f(Z, W))$. The functional expression returned on Y may be entirely sufficient for binding an essential argument in some other literal $\text{pred1}(Y, V)$. Thus Y in $\text{pred}(X, Y)$ can be marked as "out", though with a warning: if Y is further used by $\text{pred1}(Y, V)$ to determine the "out" status of V then we must remember that the functional expression binding Y contains a possibly still unbound variable in it. For example, if Y is bound to $f(\text{const}, Z)$, where Z is unbound, then unifying $\text{pred1}(Y, V)$ with $\text{pred1}(f(\text{const}, Z), Z)$ does not make the variable V "out"! On the other hand, after unifying with $\text{pred1}(f(\text{const}, Z), \text{const})$, we get V "out" all right. There are also situations when none of the above extremes would happen: for example we may unify with $\text{pred1}(f(\text{const1}, f(\text{const2}, Z)), f(\text{const2}, Z))$, obtaining a partial "out" binding for V. In general, we may introduce the notion of the *degree* of binding, defined as follows. A variable X in $\text{pred}_0(\dots X \dots)$ is "out" bound to the degree n, where n is a non-negative integer, if n is the length of the shortest path of calls $\text{pred}_0(\dots X_0 \dots)$, $\text{pred}_1(X_0, X_1)$, $\text{pred}_2(X_1, X_2)$, ..., $\text{pred}_n(X_{n-1}, X_n)$, such that X_{i-1} is "in", and X_i is partially "out" in pred_i , except for X_n which is not "out" in pred_n . If no such shortest path exists then X is fully bound with respect to the given program. A variable bound to the degree 0 is unbound. The partial "out" status of some arguments is important in analysing some literals, including recursive ones. For example, the noun literal discussed above will have N1 partially "out" when N is unbound and P is "in". As another example consider the list concatenation $@(X, Y, Z)$ implemented as

$@([], L, L)$.

$@([X|L1], L2, [X|L3]) :- @(L1, L2, L3)$.

Here $@(X, Y, Z)$ will produce a partially bound Z when it is called with X "in" and Y unbound. This allows us to conclude that the only essential arguments of @ are either X or Z.

The revised formulation of steps (L) and (R) for computing "out" arguments in a literal is given below.

(R) An argument X of literal $\text{pred}(\dots X \dots)$ on the rhs of a clause C is "out" if it is "in" in $\text{pred}(\dots X \dots)$, or the argument Y unifiable with X in the lhs of all clauses with the head literal $\text{pred}(\dots Y \dots)$ is either "in", or "out", or partially "out" as $f(Z)$ and there is a literal $\text{pred1}(\dots Z1 \dots)$ following $\text{pred}(\dots X \dots)$ in the rhs of C such that Z1 unifies with Z and Z1 is "out" in pred1 .

(L) An argument X of literal $\text{pred}(\dots X \dots)$ on the lhs of a clause is "out" if it is "in" in $\text{pred}(\dots X \dots)$, or it is "out" on the rhs of this clause (again, we must take provisions to avoid infinite descend, c.f. (D) in "in" algorithm). In addition, we ignore all clauses that contain an explicit fail literal among their rhs literals.

(L') An argument X in $\text{pred}(\dots X \dots)$ on the lhs of a clause C is partially "out" if it is bound by a functional expression $f(Z_1, \dots, Z_n)$, where some of Z_i may contain unbound variables.

3.2.3. Computing generalized minimal sets of essential arguments

The collection of minimal sets of essential arguments (MSEA's) a predicate may have depends, obviously, upon the way this predicate is defined. More clearly, if there are several alternative ways of defining a given predicate (assuming that these definitions give it the same "meaning") then we may receive possibly different collections of MSEA's each time. Still more to the point: if we alter the ordering of the rhs literals in the definition of a predicate, we may also change its set of MSEA's. We call the set of MSEA's existing for a current definition of a predicate the set of *active* MSEA's for this predicate. When we reverse the predicate we want a specific MSEA to be among its active MSEA's. If this is not already the case, then we have to alter the definition of this predicate to make the said MSEA an active one. As an example take the following clause from our Prolog parser:

```
objectbe(O1,O2,P1,P2,PSA,P) :-
    venpass(O1,O2,P1,P3),
    @([P2,P3],PSA,P).
```

Assuming that $\{O1\}$ and $\{P3\}$ are MSEA's of `venpass` and that $P3$ is "out" in `venpass` whenever $O1$ is "in", we obtain that $\{O1\}$ is the only active MSEA of `objectbe`. However, if we reverse the order of `venpass` and `@`, then $\{P\}$ becomes the new active MSEA, while $\{O1\}$ is no longer active.

Consider the following abstract clause defining predicate R_i :

$$R_i(X_1, \dots, X_i) :- Q_1(\dots), Q_2(\dots), \dots, Q_n(\dots). \quad (D1)$$

Suppose that, as defined by (D1), R_i has the set $MS_i = \{m_1, \dots, m_j\}$ of MSEA's. We call MS_i the set of active MSEA's, as opposed to the set $MR_i \supseteq MS_i$ of all MSEA for R_i that can be obtained by permuting the order of literals on the right-hand side of D1. Let us assume now that R_i occurs on rhs of some other clause, as shown below:

$$P(X_1, \dots, X_n) :- R_1(X_{1,1}, \dots, X_{1,k_1}), R_2(X_{2,1}, \dots, X_{2,k_2}), \dots, R_s(X_{s,1}, \dots, X_{s,k_s}). \quad (C1)$$

We want to compute MS , the set of active MSEA's for P , as defined by (C1), where $s \geq 0$, assuming that we know the sets of active MSEA for each R_i on the rhs.⁵ If $s=0$, that is P has no rhs in its definition, then if $P(X_1, \dots, X_n)$ is a call to P on the rhs of some clause and X^* is a subset of $\{X_1, \dots, X_n\}$ then X^* is a MSEA in P if X^* is the smallest set such that all arguments in X^* consistently unify with the corresponding arguments in at most 1 occurrence of P on the lhs anywhere in the program.⁶ In one special case, if there is only one assertion $P(X_1, \dots, X_n)$ in the entire program and every $\{X_i\}$ is a MSEA, then P 's set of MSEA's can be replaced by the set containing only one element: the empty set, i.e., by $\{\emptyset\}$. For example, the lexicon lookup predicate `vrbl(V1,V2,Num,P)` has two sets of essential arguments: $\{V1\}$ and $\{Num,P\}$. This is because these are minimal sets such that whenever we bind all arguments in either of them the resulting instantiation of `vrbl` can be unified with at most one assertion in the lexicon, which consists, among others, of the following clauses:

```
vrbl([looks|V],V,sg,look).
vrbl([look|V],V,pl,look).
vrbl([arrives|V],V,sg,arrive).
vrbl([arrive|V],V,pl,arrive).
...
```

When $s \geq 1$, that is, P has at least one literal on the rhs, we use the recursive procedure `MSEAS` to compute the set of MSEA's for P , providing that we already know the set of MSEA's for each literal occurring on the rhs. Before presenting the procedure, we introduce two auxiliary concepts. Let T be a set of terms, that is, variables and functional expressions, then $\text{VAR}(T)$ is the set of all variables occurring in the terms of T . Thus

⁵ MSEA's of basic predicates, such as `concat`, are assumed to be known apriori; MSEA's for recursive predicates are first computed from non-recursive clauses.

⁶ The *at most 1* requirement is the strictest possible, and it can be relaxed to *at most n* in specific applications. The choice of n may depend upon the nature of the input language being processed (it may be n -degree ambiguous), and/or the cost of backing up from unsuccessful calls.

$\text{VAR}(\{f(X), Y, g(c, f(Z), X)\}) = \{X, Y, Z\}$. We assume that symbols X_i in definitions (C1) and (D1) above represent terms, not just variables. For the purpose of this algorithm we also introduce the directed relation *is always unifiable with* between argument terms, and define it as follows. An argument Y is always unifiable with an argument X if they unify regardless of the possible bindings of any variables occurring in Y (after standardizing apart any unbound variables remaining in them). All variables occurring in X are unbound. Thus, any term is always unifiable with a variable; a functional expression $f(A)$ is always unifiable with a functional expression $f(B)$ if A is always unifiable with B ; a constant is always unifiable with identical constant. However, a variable is not always unifiable with a non-variable, including a functional expression. To see it, consider the variable X and functional term $f(Y)$: X is not always unifiable with $f(Y)$ because if we substitute $g(Z)$ for X then the so obtained terms do not unify. The reason for introducing this relation will be explained shortly.

The following algorithm is suggested for computing sets of active MSEA's in P : $\text{MSEAS}(\text{MS}, \text{MSEA}, \text{VP}, i, \text{OUT})$, where $i \geq 1$. The steps (3) and (7) are included for the stronger version of the algorithm that can properly handle recursive clauses.

- (1) Start with $\text{MSEA} = \emptyset$, $\text{VP} = \text{VAR}(\{X_1, \dots, X_n\})$, $i=1$, and $\text{OUT} = \text{OUT}_0 = \emptyset$. When the computation is completed, MS is bound to the set of active MSEA's for P .
- (2) Let MR_1 be the set of active MSEA's of R_1 , and let MRU_1 be obtained from MR_1 by replacing all variables in each member of MR_1 by their corresponding actual arguments of R_1 on the rhs of (C1).
- (3) *Strong version only*: If $R_1 = P$ then for every $m_{1,k} \in \text{MRU}_1$ if every argument $Y_i \in m_{1,k}$ is *always unifiable* with its corresponding argument X_i in P then remove $m_{1,k}$ from MRU_1 .
- (4) For each $m_{1,j} \in \text{MRU}_1$ ($j = 1, \dots, r$) the set $\mu_{1,j} = (\text{VAR}(m_{1,j}) - \text{OUT}_0) \cap \text{VP}$ is a subset of an active MSEA for P but only if $\phi(\mu_{1,j})$ holds, where $\phi(\mu_{1,j}) = [\mu_{1,j} \neq \emptyset \text{ or } (\mu_{1,j} = \emptyset \text{ and } \text{VAR}(m_{1,j}) = \emptyset)]$. Let $\text{MP}_1 = \{\mu_{1,j} \mid \phi(\mu_{1,j}), j=1, \dots, r\}$, where $r > 0$. If $\text{MP}_1 = \emptyset$ then (C1) is ill-formed and cannot be executed.
- (5) For each $\mu_{1,j} \in \text{MP}_1$ we do the following: (a) assume that $\mu_{1,j}$ is "in" in R_1 ; (b) compute set OUT_j of "out" arguments for R_1 ; (c) compute $\text{OUT}_{1,j} := \text{OUT}_j \cup \text{OUT}_0$; (d) recursively call $\text{MSEAS}(\text{MS}_{1,j}, \mu_{1,j}, \text{VP}, 2, \text{OUT}_{1,j})$; (e) assign $\text{MS} := \bigcup_{j=1..r} \text{MS}_{1,j}$.
- (6) In some i -th step, where $1 \leq i \leq s$, and $\text{MSEA} = \mu_{i-1,k}$, let's suppose that MR_i and MRU_i are the sets of active MSEA's and their instantiations with actual arguments of R_i for the literal R_i on the rhs of (C1).
- (7) *Strong version only*: If $R_i = P$ then for every $m_{i,u} \in \text{MRU}_i$ if every argument $Y_i \in m_{i,u}$ is *always unifiable* with its corresponding argument X_i in P then remove $m_{i,u}$ from MRU_i .
- (8) Again, we compute the set $\text{MP}_i = \{\mu_{i,j} \mid j=1, \dots, r_i\}$, where $\mu_{i,j} = (\text{VAR}(m_{i,j}) - \text{OUT}_{i-1,k})$, where $\text{OUT}_{i-1,k}$ is the set of all "out" arguments in literals R_1 to R_{i-1} .
- (9) For every $m_{i,j}, m_{i,k} \in \text{MRU}_i$, if $m_{i,j}, m_{i,k}$ are obtained from the MSEA's for R_i that are active at the same time (i.e., belong to the set of active MSEA's for the same definition of R_i), and such that $\mu_{i,j} \subseteq \mu_{i,k}$, then eliminate $\mu_{i,j}$ from MP_i . (When we compute active MSEA's only as we do here, then all $m_{i,j}$ are active at the same time; but see below for a possible extension to this algorithm.)
- (10) For each $\mu_{i,j}$ remaining in MP_i where $i \leq s$ do the following:
 - (a) if $\mu_{i,j} = \emptyset$ then: (i) compute set OUT_j of "out" arguments of R_i ; (ii) compute $\text{OUT}_{i,j} := \text{OUT}_j \cup \text{OUT}_{i-1,k}$; (iii) call $\text{MSEAS}(\text{MS}_{i,j}, \mu_{i-1,k}, \text{VP}, i+1, \text{OUT}_{i,j})$;
 - (b) otherwise, if $\mu_{i,j} \neq \emptyset$ and $\mu_{i,j} \subseteq \text{VP}$ then: (i) assume $v = \mu_{i,j} \cap \text{VP}$ is "in"; (ii) compute OUT_j of "out" arguments of R_i ; (iii) compute $\text{OUT}_{i,j} := \text{OUT}_j \cup \text{OUT}_{i-1,k}$; (iv) call $\text{MSEAS}(\text{MS}_{i,j}, \mu_{i-1,k} \cup v, \text{VP}, i+1, \text{OUT}_{i,j})$;
 - (c) otherwise quit returning nil.
- (11) Compute $\text{MS} := \bigcup_{j=1..r} \text{MS}_{i,j}$;
- (12) For $i=s+1$, i.e., for $\text{MSEAS}(\text{MS}, \text{MSEA}, \text{VP}, s+1, \text{OUT})$, do $\text{MS} := \{\text{MSEA}\}$.

The weak version of this algorithm, that is, with steps (3) and (7) removed, is sufficient in a great majority of cases, but it cannot properly handle certain types of recursive definitions. Consider, for example, the problem of assigning

the set of MSEA's to $\text{mem}(\text{Elem}, \text{List})$, where mem (list membership) is defined as follows:

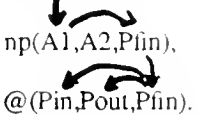
```
mem(X, [X|L]).
mem(X, [_|L]) :- mem(X, L).
```

The weak version of MSEAS assigns $\text{MS} = \{\emptyset\}$, which means that mem can be evaluated successfully at any time. This is true, but impractical since there are infinite number of lists of which a certain element is a member. The strong version of MSEAS eliminates $\{\text{Elem}\}$ as a possible MSEA because the X in mem on the rhs is always unifiable with the X in mem on the lhs of the recursive second clause. This leaves us with only one MSEA, which is $\{\text{List}\}$.

The algorithm presented here can also be used to compute the set of all MSEA's for P by repeating all the steps for every permutation of literals on the rhs of (C1) and every combination of active MSEA's for R_i 's. It can also be modified to do so by taking MR_i to be the set of all MSEA's for R_i and repeating all the steps of the algorithm for all permutations of R_i 's on the rhs of (C1). To obtain a meaningful result, MSEA's in MR_i 's must be grouped into sets of these which are active at the same time, i.e., they belong to the set of active MSEA's for a specific definition of P . MSEA's belonging to different groups give rise to alternative sets of MSEA's in the final set MS . Note that in this modified algorithm, MS becomes a set of sets of sets.

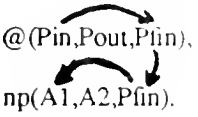
As an example, we compute the set of all MSEA's for the following simple clause.⁷ For the sake of simplicity we assume that MSEA's of the literals on the rhs are known.

```
quack(A1, A2, Pin, Pout) :-
    np(A1, A2, Pfin),
    @(Pin, Pout, Pfin).
```



If $\{A1\}$ is the only active MSEA in np , and Pfin is "out" whenever $A1$ is "in", then $\{A1\}$ is a MSEA in quack . This may be easily seen by drawing a directed arc from $A1$ to Pfin (indicating "in"-"out" transfer) and from Pfin in np to Pout in $@$ (indicating "out"-"in" transfer). Because $\{\text{Pfin}\}$ is an active MSEA in $@$ ⁸, our job is done. On the other hand, if $\{\text{Pfin}\}$ were the only active MSEA in np , then the resulting MSEA for quack would be $\{A1, \text{Pin}\}$. Next, we reverse the order of np and $@$, obtaining the following new definition of quack :

```
quack(A1, A2, Pin, Pout) :-
    @(Pin, Pout, Pfin),
    np(A1, A2, Pfin).
```



Following the same steps as above we obtain that $\{\text{Pin}\}$ is an active MSEA in quack , assuming that $\{\text{Pfin}\}$ is an active MSEA in np . If $\{A1\}$ were the only active MSEA in np , as before, then we would obtain $\{A1, \text{Pin}\}$ as MSEA for quack . If this MSEA proves unacceptable (because, say, $A1$ is not expected to be bound when calling quack) then we need to change the definition of np so that Pfin becomes its active MSEA. Again we draw directed arcs from Pin to Pfin in $@$ and from Pfin in $@$ to Pout in np . We may note here that since Pout is not essential, quack cannot be reversed to compute $A1$ out of Pout . The only way to do this would be to change the definition of $@$ (list concatenation), but this we disallow treating $@$ as atomic primitive. The same result can be obtained using the modified version of the algorithm in which all MSEA's (active or not) of the literals on the rhs of a clause are examined at the same time.

Let now m be a MSEA of P such that $m \in \text{MP} - \text{MS}$, i.e., m is not an active MSEA, where MP is the set of all MSEA's, and MS is the set of all active MSEA's for P . If we want to execute P successfully with only the arguments in m being "in", we have to reorder rhs of (C1) so that m becomes an active MSEA. If m is a non-active MSEA then we may actually be able to accomplish this; otherwise it is impossible without rewriting the program. Suppose then, as we did above, that m is a MSEA for P , though it isn't active at the moment. Thus we reorder the literals on the rhs of (C1) so that m becomes active, but, unfortunately, while doing so we had to place R_i in such a

⁷ This example was suggested by Ralph Grishman

⁸ Concatenation ($@$) has two active MSEA's, here, $\{\text{Pin}\}$ and $\{\text{Pfin}\}$.

position that none of its active MSEA's is guaranteed to be "in", and only some $mr \in MR_1 - MS_1$ is so guaranteed. Technically then such a reordering of (C1)'s rhs is not admissible, because R_1 cannot be executed successfully with only mr being "in". In this situation we may decide to replace the current definition of R_1 , which is (D1), by a new definition (D2) in which the literals on the rhs of (D1) are reordered so that mr becomes active. Basically, there are two ways to approach the problem of goal reordering on the right-hand side of P:

- (1) Place R_1 on the rhs as soon as one of its active MSEA's is guaranteed to be "in"; otherwise wait and place other literals ahead (if possible). Change definition of R_1 only when no further progress is possible.
- (2) Place R_1 on the rhs as soon as any of its MSEA's (active or not) is "in". Change definition of R_1 accordingly.

What happens, however, if R_1 is on the rhs of a definition of some other predicate P1? If we alter the definition of R_1 to make the new definition of P admissible, we may end up with having to change it again to have both P and P1 admissible. In other words, if MR_1 is the set of MSEA for R_1 that guarantees admissibility of P (as far as R_1 is concerned), and MR_j is the set of MSEA for R_1 that would make P1 admissible then we have to change the definition of R_1 so that at least one MSEA in $MR_1 \cap MR_j$ becomes active. This can get further complicated by possible changes to the definitions of other predicates on the rhs of P and P1. The best way to avoid such complications is to limit the changes to the definition of P1 itself, and also assume that certain primitive predicates (such as concat, member, cons, etc.) are not subject to re-definition and must be considered atomic.

3.2.4. A detailed example

Before we proceed, let us analyse in detail the process of computing the set of MSEA's for a simple recursive literal. Let's assume the literal vp is defined as follows:

- [1] $vp(V1,V3,Args,Vsem) :- vp(V1,V2,[Csem|Args],Vsem),np(V2,V3,Csem).$
- [2] $vp(V1,V2,Args,Vsem) :- v(V1,V2,Args,Vsem).$
- [3] $v([loves|X],X,[Obj,Subj],love(Subj,Obj)).$
- [4] $np([mary|X],X,mary).$

For the sake of simplicity, we assume that MSEA's of v and np are already known and are $\{(V1,Args),(Vsem)\}$ and $\{(V2),(Csem)\}$, respectively. In other words, both literals have two alternative sets of MSEA. We include $Args$ among the essential arguments of v , even if it may not be required especially when each verb is assumed to translate into a single, unique predicate.⁹ We proceed to compute the set of MSEA's for vp in [1] using first the weak version of the procedure MSEAS described in the previous section, i.e., leaving out steps (3) and (7). The first step is to create the set VP of variables of the head literal. We obtain $VP = \{V1,V3,Args,Vsem\}$. Next, we have to compute the set of MSEA's for the first literal on the right-hand side, which happens to be vp as well. To accomplish this we use the second clause [2], and obtain the result $\{(V1,Args),(Vsem)\}$. This is because in [2] vp rewrites to v and MSEA's of v are known. From here we obtain that $MRU_1 = \{(V1,Args),(Vsem)\}$, and subsequently that $\mu_{1,1} = \{V1,Args\}$ and $\mu_{1,2} = \{Vsem\}$. They both satisfy the condition ϕ , and thus we have $MP_1 = \{(V1,Args),(Vsem)\}$. The next step is to call the procedure recursively for each element of MP_1 , thus creating alternative branches of computation. Let's concentrate on $\mu_{1,1}$ first, and assume its elements are "in" arguments in vp on the right-hand side of [1]. We proceed to compute the set of "out" arguments in this vp literal, assuming that $V1$ and $Args$ are "in". The result is $OUT_1 = \{V2,Vsem\}$, which follows from the fact that vp rewrites to v . Since OUT_0 is empty, we obtain $OUT_{1,1} = OUT_1 = \{V2,Vsem\}$. Now we call MSEAS recursively with its second parameter bound by $\{(V1,Args)\}$. Within this call, the above steps are repeated for the second literal on the rhs of [1], which is np . MSEA's for np give rise to $MRU_2 = \{(V2),(Csem)\}$, from where we compute $MP_2 = \{(V2),(Csem)\}$. We take $\mu_{2,1} = \{V2\}$ as the set of "in" arguments in np , obtaining $OUT_{2,1} = \{V3,Csem\}$. After one more, this time trivial, recursive call to MSEAS we obtain $MS = \{(V1,Args)\}$. The other element of MP_2 is $\mu_{2,2} = \{Csem\}$ but this one is not considered because $\{Csem\}$ is not a subset of VP . This concludes exploration of the recursive branch below $\mu_{1,1}$. For $\mu_{1,2} = \{Vsem\}$ we assume $Vsem$ to be "in" and compute $OUT_{1,2} = \{V1,Csem,Args\}$. Here $V1$ is only partially "out" but this will suffice. $Csem$ and $Args$ are fully "out" since we know $Vsem$, and vp rewrites to v where the third

⁹ Some verbs can be both transitive and intransitive depending upon use. For example, *read* can take either one argument, as in *John reads*, or two arguments, as in *John reads a book*.

argument is always "out" whenever the last argument is "in". Thus, we obtain $MP_2 = \{\{V2\}, \{\}\}$. The first element of this set is not a subset of VP, and thus is not taken under consideration. The other yields $MS = \{\{Vsem\}\}$ in the next call to MSEAS. In the end we obtain two alternative sets of MSEA for vp: $\{V1, Args\}$ and $\{Vsem\}$. This indicates that vp could, in principle, be run in two directions: with V1 and Args known, or else with Vsem known. The predicate is thus reversible even without changing its definition. This result is based on the assumption that the recursion in clause [1] will not go forever, and that the stop condition in [2] will be used successfully. The reader may note that this assumption is not valid for the code consisting of clauses [1] to [4] but it could be achieved by normalizing the program as discussed in section 4. The fact that this program is actually ill-formed as it stands falls out neatly if we use the stronger version of the MSEAS procedure, i.e., we replace back the steps (3) and (7). Both MSEA's $\{V1, Args\}$ and $\{Vsem\}$ obtained from the non-recursive clause for vp are eliminated in the analysis of the recursive clause because V1 and Vsem are passed unchanged from the left to the right side, and Args rewrites into a functional expression which is always unifiable with it. We obtain $M_{1,2}^2 = \emptyset$ in step (4), and thus that the recursive clause is ill-formed.

At the end it may be useful to note that in the example analysed here we knew the set of MSEA's for vp from clause [2], from the weak version of MSEAS, even before considering the recursive rule in [1]. This will not always be the case, and the set obtained from non-recursive rules could be narrowed to a smaller set of MSEA's.¹⁰ To see that consider a modification of clause [1], where variable Csem is replaced by some new variable Xsem that does not occur in np. Since now Csem would not be "out" after vp returns in [1], $\mu_{2,2}$ would not be empty but would contain Csem, which is not in VP. This would cut off the entire branch under $\mu_{1,2}$, and thus MSEAS (the weak version) would return only one possible MSEA for vp, that is $\{V1, Args\}$.

3.2.5. Reordering literals on clause's rhs

When attempting to expand a literal on the rhs of any clause the following basic rule should be observed: never expand a literal before at least one its MSEA's is "in", which means that all arguments in at least one MSEA are bound. If the "in" MSEA is not active then recursively reorder the rhs of the definition of the predicate in question. The following algorithm uses this simple principle to reorder rhs of parser clauses for reversed use in generation. This algorithm uses the information about "in" and "out" arguments as computed by the algorithm given in the section above.

Reordering rhs of a clause: head :- old-rhs.

```

REORDER("head :- old-rhs", "head :- new-rhs");
begin
  new-rhs :=  $\emptyset$ ;
  compute and mark "in" arguments in head;
  repeat until old-rhs =  $\emptyset$ 
    begin
      mark as "in" those arguments in old-rhs which are either
        marked "in" in head, or
        marked "in" or "out" in new-rhs, or
        constants;
      find the left-most literal L in old-rhs such that
        it has an "in" MSEA;
      if no such literal found then quit("not reversible");
      compute and mark "in" and "out" arguments in L;
      new-rhs := APPEND-AT-THE-END(new-rhs, L);
      old-rhs := REMOVE(old-rhs, L);
    end;
  mark as "out" those arguments in head that are marked "out" in new-rhs
end;
```

¹⁰ An empty set of MSEA indicates an ill-formed clause.

As discussed in the previous section this simple algorithm can be generalized to include complicated re-definition of predicates of certain literals so that they can be scheduled even if none of its currently active MSEA's is "in". In practice, we proceed top-down altering definitions of predicates of the literals to make their MSEA's active as necessary. When reversing our parser, we start with the top level predicate parse(S,P) assuming that its only active MSEA to be {P}, where P is bound to the regularized parse structure of a sentence. We explicitly identify and mark P as "in" and add the requirement that S must be marked "out" upon completion of rhs reordering. This last condition is necessary if the reversed program is to retain its intended meaningfulness, that is, to produce the expected output. We proceed to adjust the definition of parse (which is originally written with {S} as the active MSEA) to reflect that now {P} is active. We continue until we reach the level of atomic or non-reversible primitives such as concat, member, or dictionary look-up routines. If this top-down process succeeds at reversing predicate definitions at each level down to the primitives, and the primitives need no re-definition, then the process is successful, and the reversed-parser generator is obtained.

The revised version of REORDER is given below as INVERSE(clause,ins,outs), where clause is in the form head :- old-rhs, ins is the set of these arguments in VAR(head) that are known to be "in", while outs are those arguments in VAR(head) that are required to be "out" after the clause is executed.

Inverting the Prolog program with the top-level clause head :- old-rhs.

```

INVERSE("head :- old-rhs",ins,outs):
begin
  compute M the set of all MSEA's for head;
  for every MSEA m  $\in$  M do
    begin
      OUT :=  $\emptyset$ ;
      if m is an active MSEA such that  $m \subseteq \text{ins}$  then
        begin
          compute and mark "out" arguments in head; add them to OUT;
          if  $\text{outs} \subseteq \text{OUT}$  then DONE("head:-old-rhs")
        end
      else if m is a non-active MSEA such that  $m \subseteq \text{ins}$  then
        begin
          new-rhs :=  $\emptyset$ ; old-rhs-1 := old-rhs; QUIT := false;
          for every literal L in the program do  $M_L := \emptyset$ ;
          {this is done only once during the inversion process}
          repeat
            mark as "in" those arguments in old-rhs-1 which are either
              constants, or marked "in" in head, or
              marked "in" or "out" in new-rhs;
            LL: find the left-most literal L in old-rhs-1 such that it has an "in" MSEA  $m_L$ ;
            if L exists then
              begin
                if  $m_L$  is non-active in L then
                  if  $M_L = \emptyset$  or  $m_L \in M_L$  then
                    begin
                       $M_L := M_L \cup \{m_L\}$ ;
                      for every clause "L1 :- rhsL1" in the program
                        such that L1 has the same predicate as L do
                          INVERSE("L1 :- rhsL1", $M_L$ , $\emptyset$ )
                    end
                  else try another L at LL;
                compute and mark "in" and "out" arguments in L;
                add "out" arguments to OUT;
                new-rhs := APPEND-AT-THE-END(new-rhs,L);
                old-rhs-1 := REMOVE(old-rhs-1,L)
              end {if}
          until QUIT
        end
    end
  end

```

```

        else QUIT := true
        until old-rhs-1 =  $\emptyset$  or QUIT;
        if outs $\subseteq$ OUT then DONE("head:-new-rhs")
    end {elseif}
end; {for}
write("program cannot be inverted as specified")
end;

```

The reader may note that INVERSE is not a complete program yet, and it needs a few minor adjustments before it can be safely used. For one thing, we have to make sure that the recursive calls do not create an infinite loop. This may happen if the literal selected for recursive inversion on the right-hand side of the currently open clause has the predicate which is identical to that in the head of this clause, and this same clause is again selected for inversion. In a more general case we have to block recursive inversion of any clause which is being currently open for inversion by the chain of recursive calls to INVERSE. This can be accomplished by keeping a list of all clauses open for inversion and testing its membership before every recursive call to INVERSE. Another problem is that, as it stands, INVERSE cannot truly guarantee that, in reaching a particular ordering of the literals on the right-hand side of a clause, it will accomodate more than one active MSEA at a time. What it does at present is merely to make sure that an inverted literal is not inverted again. On the other hand, making a literal executable in more than one directions at the same time, rather than, what INVERSE does, only changing its direction from one to another, creates a somewhat different problem, which may not be relevant in the present context. Nonetheless, a more complete algorithm that would include the above provision is not difficult to obtain and we leave it for the reader as an exercise.

3.2.6. Essential arguments for greater efficiency

Even though certain arguments in a literal may not be strictly essential, in the sense that the literal will execute successfully without them being bound, the efficiency of the program can be increased if these arguments are actually "in" before the said literal is executed. As an example, we may consider the following (hypothetical) parser clause:

```

yesnoq(A1,A4,P) :-
    verb(A1,A2,Num,Gen,P2),
    subject(A2,A3,Num,Gen,P1),
    object(A3,A4,P1,P2,P).

```

When rewriting this clause for generation, we would place object first (it has P "in", and A3, P1, P2 "out"), but then the REORDER algorithm preserves the order of the original definition if more than one literal becomes available for scheduling at the same time. Thus verb would be scheduled second and subject last, resulting in the following generator clause:

```

yesnoq(A1,A4,P) :-
    object(A3,A4,P1,P2,P),
    verb(A1,A2,Num,Gen,P2),
    subject(A2,A3,Num,Gen,P1).

```

Suppose now that the information about the number and gender agreement of the pair verb-subject is placed in the translation P1 of the subject, but not in the translation P2 of the verb. What happens is that when we call verb with P2 "in" but the rest of its arguments unbound, we pick up a random lexical form of P2 (which contains verb's root form) from the lexicon and later, while executing subject, we try to match it against the number/gender information contained in P1. If we are unlucky, we may backup several times before these values coincide. A better approach would be to schedule subject first, thus setting Num and Gen "in" and then make one sure call to verb. What we need to do now is to guarantee that the reordering algorithm gets things in the right order! This could be easily achieved if we make Num and Gen essential in verb together with P2. In other words, we want the set of MSEA's for verb to be $\{\{A1\},\{Num,Gen,P2\}\}$, which is quite understandable, as Num, Gen and P2 (the root form of the verb) fully define the verb's lexical form. When verb is called with A1 and either Num or Gen unbound, the number of different lexical verb forms that can be matched in the lexicon is almost always greater than one, and we may have to make a guess. This consideration alone can be a sufficient criterion to make Num and Gen essential. Adding Num and Gen to the MSEA for verb results in the following generator clause:

```

yesnoq(A1,A4,P) :-
    object(A3,A4,P1,P2,P),

```

```

subject(A2,A3,Num,Gen,P1),
verb(A1,A2,Num,Gen,P2).

```

The method can also be used to determine other essential arguments of lexicon look-up primitives, and other fact clauses. Thus, `noun(N1,N2,Num,P)` has `{N1}` and `{P}` as MSEA's simply because these are only arguments that can reduce number of possible matches in the lexicon from a potentially large number to 1 (if `N1` is "in"), or to 2 (if `P` is "in"). If we add `Num` to the second MSEA, obtaining `{Num,P}`, then, here too, the number of possible matches in the lexicon is reduced to 1.

This method of expanding the set of essential arguments can be used to achieve a marked improvement in the performance of the generator, but the method may easily run into trouble if the parser is not uniformly designed. For example, if the translation `P1` of `subject` contains information about its gender only and the translation of `verb` contains information about the number only, then extending sets of essential arguments of `subject` to `{Num,P1}` and of `verb` to `{Gen,P2}` would result, we might imagine, in a deadlock situation in which both wait for each other forever. INVERSE will not deadlock, but it will refuse to reverse such a program. However, it may sometimes be more desirable to detect and break such potential deadlocks rather than just quit. This effect may be achieved by making `Num` and `Gen` second-class essential arguments, and thus allowing for INVERSE to override their status if it cannot reorder a clause otherwise. In such situations we may be confronted with the problem of choosing lesser of two evils. Which way do we fare better: by executing `subject` first, or by executing `verb` first? The answer seems to depend on the number of guesses each of them would have to make, and favoring the one which makes fewer guesses. On the other hand, this decision must also depend on how much computation has to be undone and redone in case we make a wrong guess; in other words we have to calculate the average (or worst) amount of computation (and time) wasted in each case. This, however, may not be easy to determine. An obvious alternative solution to the problem of deadlocked goals, apart from their concurrent evaluation, is to normalize the grammar so that the deadlock-prone clauses are eliminated, or at least their number minimized. We return to this problem in section 4.

3.3. An example of inverting a literal

In this section we go step by step through the process of literal inversion. Let's consider the following parser clause¹¹:

```

assertion(A1,A41,WHQ,P) :-
    sa(A1,A11,PA1),
    subject(A11,A2,WHQ,Num,P1),
    sa(A2,A21,PA2),
    verb(A21,A3,Sts,Num,P2),
    sas(A3,A31,Sts,PA3),
    object(A31,A4,O,Sts,assert,P1,P2,PSA,P),
    sao(A4,A41,O,PA4),
    #([PA1,PA2,PA3,PA4],PSA).

```

When used for parsing, `A1` and `WHQ` are "in" while `A41` and `P` are "out" in the lhs literal assertion. On the rhs, object literal, for example, has `A31`, `Sts`, `assert`, `P1` and `P2` "in", and `A4`, `O` and `P` "out".¹² `PSA` is neither "in" nor "out". When the clause is used for generation, then `P` is "in" and `A1` is "out". In object literal we have `P` and `assert` "in", and `A31`, `O`, `P1`, `P2` and `PSA` "out". `A4` and `Sts` are neither "in" nor "out".

We call INVERSE with the above clause as the value of the first parameter, and `ins={P}`, and `outs={A1}`. The set of MSEA's for the head literal `assertion` is `{{A1},{P}}`. As it stands, `{A1}` is an active MSEA in assertion, while `{P}` is not. We need to invert assertion so that `{P}` becomes active and thus guarantees execution of assertion with argument `P` "in". Since `{A1}` is not a subset of `ins`, we proceed to consider the only remaining, non-active MSEA, i.e., `{P}`. We set up `new-rhs=∅`, `old-rhs-1=old-rhs`, and `QUIT=false`. Next, we mark "in" arguments in `old-rhs-1`. Since `new-rhs` is empty, we look for those which are marked "in" in the head literal, obtaining `P` in object. In addition, the fifth argument of object literal is a constant. Since `{P}` is a MSEA for object (and let's assume that it is

¹¹ For the sake of simplicity, in this, and in the following examples, we may occasionally omit some of the arguments to selected literals, as compared with the code given in appendices A and B, if their presence is not essential to illustrate the problem at hand.

¹² We may note here that `P` in object is only partially "out" because it is being constructed out of `P1`, `P2` and `PSA` while the latter is still unbound. However, `P` is fully "out" in assertion because `PSA` is "out" in #.

an active MSEA, for simplicity) we compute that P is "in" and A31, O, P1, P2 and PSA are "out" in *object*. We thus obtain that OUT={A31,O,P1,P2,PSA} and that

```
new-rhs =
  object(A31,A4,O,Sts,assert,P1,P2,PSA,P)
old-rhs-1 =
  sa(A1,A11,PA1),
  subject(A11,A2,WHQ,Num,P1),
  sa(A2,A21,PA2),
  verb(A21,A3,Sts,Num,P2),
  sas(A3,A31,Sts,PA3),
  sao(A4,A41,O,PA4),
  #([PA1,PA2,PA3,PA4],PSA)
```

In the next turn of the repeat loop, we mark the following "in" arguments in old-rhs-1: P1 in subject, P2 in verb, A31 in sas, O in sao and PSA in #. This is because all these arguments are now "out" in new-rhs. The left-most literal with an "in" MSEA in old-rhs-1 is subject. When {P1} is "in", then "out" arguments in subject are A11 and Num. We add them to the set OUT, obtaining OUT = {A11,A31,Num,O,P1,P2,PSA}. Next, we modify new-rhs and old-rhs-1 as follows:

```
new-rhs =
  object(A31,A4,O,Sts,assert,P1,P2,PSA,P),
  subject(A11,A2,WHQ,Num,P1)
old-rhs-1 =
  sa(A1,A11,PA1),
  sa(A2,A21,PA2),
  verb(A21,A3,Sts,Num,P2),
  sas(A3,A31,Sts,PA3),
  sao(A4,A41,O,PA4),
  #([PA1,PA2,PA3,PA4],PSA)
```

Now the "in" arguments in old-rhs-1 are: A11 in the first sa, P2 and Num in verb, A31 in sas, O in sao and PSA in #. {P2,Num} is a MSEA for verb (again, assume it is active), which makes A21 and Sts "out" in verb. We add these two to OUT, obtaining OUT = {A11,A21,A31,Num,O,P1,P2,PSA,Sts}. The new values of new-rhs and old-rhs-1 are as follows:

```
new-rhs =
  object(A31,A4,O,Sts,assert,P1,P2,PSA,P),
  subject(A11,A2,WHQ,Num,P1),
  verb(A21,A3,Sts,Num,P2)
old-rhs-1 =
  sa(A1,A11,PA1),
  sa(A2,A21,PA2),
  sas(A3,A31,Sts,PA3),
  sao(A4,A41,O,PA4),
  #([PA1,PA2,PA3,PA4],PSA)
```

In the next step we'll add # literal at the end of new-rhs. After this, PA1, PA2, PA3 and PA4 will become marked "in" in old-rhs-1. The remaining literals in old-rhs-1 are moved to new-rhs in the same fashion, but there will be no more changes in ordering. The final effect is shown below.

```
assertion(A1,A41,WHQ,P) :-
  object(A31,A4,O,Sts,assert,P1,P2,PSA,P),
  subject(A11,A2,WHQ,Num,P1),
  verb(A21,A3,Sts,Num,P2),
  #([PA1,PA2,PA3,PA4],PSA),
  sa(A1,A11,PA1),
  sa(A2,A21,PA2),
  sas(A3,A31,Sts,PA3),
  sao(A4,A41,O,PA4).
```

4. Expanding the applicability of the inversion algorithm

For the most effective use of the inversion algorithm, the original PROLOG program of the parser must be in a certain "normal" form, that is, a form that would assure program invertibility and produce the maximum efficiency of the generator program. The most obvious necessary condition that a program must satisfy in order to be invertible by INVERSE, is that each clause in this program must be invertible. The sufficient condition is that the inverted clauses taken together create an executable whole. Thus, any program whose inversion involves manipulating of several clauses at once in order to establish some global, inter-clausal ordering of literals, is not in the "normal" form, and will not be reversed by the present version of the algorithm. Nonetheless, "non-normal" programs (or grammars) do exist, and can be quite well-formed. Moreover, they can be also reversible, though this may involve somewhat more skill. Understandably, we would like to expand the applicability of our inversion algorithm to cover these "non-normal" cases as well. One way to achieve this end is to construct a normalizer that would transform the non-normal programs into normal ones, upon which INVERSE could be run. The normalization process would replace groups of clauses in the original program by semantically equivalent new clauses which possess desired properties, that is, they are reversible by INVERSE. This operation can also be performed at the level of the grammar, so that groups of productions are replaced by new productions, in a manner somewhat similar to that of removing left-recursion. Of course, we need a method of telling a "normal" grammar or program from a "non-normal" one. This may be done on the clause by clause basis, or it may involve larger fragments of code, but the method must not rely on criteria which require the normalizer program to know whether a given clause is reversible or not, for if it did, we would have to attempt to reverse this clause first, before normalization. While such a post-normalizer is not entirely out of place, we are probably better off if we expand the coverage of INVERSE over at least these "non-normal" cases which cannot be normalized before the inversion starts. The ultimate goal is to describe a set of conditions that must be satisfied in designing the grammar so that it could be translated into an efficient parser or generator operating under a specific evaluation strategy (a top-down interpretation in our case). If a grammar does not meet these criteria, it should be possible to "normalize" it using an automated normalization. The compiler/normalizer would provide an extra degree of freedom to a linguist who needs not necessarily be aware of the evaluation requirements for the parser or generator based on his grammar.

In this section we discuss only a few specific cases when a normalization may be required, each of them being motivated by a different set of considerations. We show how these normalizations apply to the PROLOG code and discuss whether they could be made a part of the normalizer program, or whether they call for an extension to the inversion algorithm itself. In the latter case, we also suggest how this extension could be accomplished. First, we discuss how the clauses in a parser program can be "normalized" to assure meaningful inversion by INVERSE. Then we look at how the normalization process can be used to enhance efficiency of the inverted program. Although we limit our discussion to the actual PROLOG code, it will be also applicable to more abstract rewriting systems, such as DCG's.

4.1. Inverting deadlock-prone clauses

The inversion algorithm, as realized by the procedure INVERSE, requires that for each clause in the parser code we can find a definite order of literals on its right-hand side that would satisfy the requirements of running this clause in the reverse: appropriate minimal sets of essential arguments (MSEA's) are bound at the right time. However, this requirement is by no means guaranteed and INVERSE may encounter clauses for which no ordering of the literals on the right-hand side would be possible. It may happen, of course, that the clause itself is ill-formed but this is not the only situation. It may be that two or more literals on the right-hand side of a clause cannot be scheduled because each is waiting for the other to deliver the missing bindings to some essential arguments. As discussed previously, we can sometimes break such deadlocks by allowing the generator to make a limited amount of guesswork. This solution does not appear very satisfactory, though. Fortunately, much of the need for the crude deadlock breaking can be removed by applying a normalization to the grammar that would replace deadlock-prone clauses by equivalent new clauses that can be reordered for generation. The general solution to this normalization problem is still under investigation; here, we illustrate the problem and a possible solution using one specific situation.

Consider the following, somewhat abstract, parser clause:

`sent(P) :- np(Num,Pers,P1),vp(Num,Pers,P1,P).`

where string variables are omitted for clarity, and P1 and P carry the translations of np and sent, respectively. Suppose that P1 is the only essential argument in np, and that Num and Pers are "out" whenever P1 is "in". In vp, the essential arguments are {P,Num,Pers} and P1 is "out". Obviously, this clause cannot be inverted for generation, since in order to fire vp we need to know the bindings to Num and Pers, in addition to P (which is passed from sent).

These we could get by firing np, but we cannot do this either since we need to know the binding to P1, which is unavailable until vp is executed. A deadlock. Suppose, however, that vp has the following expansion:

`vp(Num,Pers,P1,f(P1,P2,P3)) :- verb(Num,Pers,P2),compl(P3).`

or, perhaps,

`vp(Num,Pers,P1,P) :- verb(Num,Pers,P2),compl(P3),combine([P1,P2,P3],P).`

As we can see, the critical information required to fire np literal in the sent clause, that is, the binding to P1, can be obtained only after a partial evaluation of vp, and before the bindings for Num and Pers become indispensable. Therefore, even though {P,Num,Pers} create a MSEA for vp, the bindings to the last two are not used until the literal verb(Num,Pers,P2) is called. Indeed, after reordering vp for generation we obtain the following clause:

`vp(Num,Pers,P1,P) :- combine([P1,P2,P3],P),verb(Num,Pers,P2),compl(P3).`

Now we can see in which order the literals should be evaluated to avoid the deadlock. We need to evaluate combine([P1,P2,P3],P) first, then np(Num,Pers,P1), then verb(Num,Pers,P2), and finally compl(P3). One way to achieve this serialization is to combine both clauses into a single new clause replacing vp literal in the sent clause by the right-hand side of the vp clause, and propagating all unifications between variables as necessary. So obtained new clause is subsequently inverted in the usual fashion. The net effect is as follows:

`sent(P) :- combine([P1,P2,P3],P),np(Num,Pers,P1),verb(Num,Pers,P2),compl(P3).`

The same effect can be achieved for the first version of the vp clause above by "lifting"¹³ the translation pattern f(P1,P2,P3) from vp to sent, obtaining the following clause:

`sent(f(P1,P2,P3)) :- np(Num,Pers,P1),verb(Num,Pers,P2),compl(P3).`

or, more conservatively,

`sent(f(P1,P2,P3)) :- np(Num,Pers,P1),vp1(Num,Pers,P2,P3).
vp1(Num,Pers,P2,P3) :- verb(Num,Pers,P2),compl(P3).`

There are basically two ways of implementing this kind of inter-clausal literal reordering. One is to make it a part of the "grammar normalization" process and thus leaving the INVERSE procedure intact. The advantage of this solution is that the inversion algorithm will retain its relative simplicity and elegance. The problem is that it may be difficult to single out the clauses that should be considered for this transformation without actually attempting to reorder their literals in the first place. This leads us to the second alternative implementation which involves modifications to INVERSE itself. The new INVERSE would not limit its operation to single clauses. Instead, upon failing to find an acceptable ordering for the literals in a clause, it would look at the clauses whose heads are involved in the current deadlock, and expand the reordering process to their right-hand sides also. In other words, we will create a new clause for each possible expansion of the literal in question. If the expanded reordering eventually yields an executable sequence, appropriate clauses will be created and used to replace those in the parser. If, on the other hand, the expansion brings no solution (we may hit the lexicon assertions) then we may have no choice but to allow some guesswork to be performed by the generator. However, in this latter case there isn't probably much we can do anyway, because this would mean that the generator does not have enough information to make deterministic selections at all. A word of caution must be added here, however. While expanding a literal on the right-hand side of a clause we must make sure that we are not going to end up in an infinite loop that might be created by expanding recursive literals. Thus a loop checking must be made while expanding. Another problem may be created by the presence of "ill-formed" recursive rules. These need to be normalized before the inversion process starts, as discussed in the following subsection.

4.2. Normalizing ill-formed recursive literals

Thus far we have tacitly assumed that the grammar on which our parser is based is written in such a way that it can be executed (for parsing) by a top-down, left-to-right interpreter, such as the one used by PROLOG. If this is not the case, that is, if the grammar requires a different kind of interpreter, then the question of invertibility can only be related to this particular type of interpreter. If we want to use the inversion algorithm described here to invert a parser written for an interpreter different than top-down and left-to-right, we need to convert it first into a version that can be executed by a top-down interpreter. This problem is not, strictly speaking, of our present concern,

¹³ We shall have some more to say about the "lifting" transformation later.

nonetheless we outline here how such situations can be dealt with within the present framework.

Consider, for example, the following situation.¹⁴

```
[01] sent(V1,V3,Sem) :- np(V1,V2,Ssem),vp(V2,V3,[Ssem],Sem).
[02] vp(V1,V3,Args,Vsem) :- vp(V1,V2,[Csem|Args],Vsem),np(V2,V3,Csem).
[03] vp(V1,V2,Args,Vsem) :- v(V1,V2,Args,Vsem).
[04] v(V1,V2,[Iobj,Obj,Subj],give(Subj,Obj,Iobj)) :- give(V1,V2).
[05] v(V1,V2,[Obj,Subj],love(Subj,Obj)) :- love(V1,V2).
[06] v(V1,V2,[Subj],arrive(Subj)) :- arrive(V1,V2).
[07] love([Iloves|X],X).
[08] love([Iloved|X],X).
[09] give([Igives|X],X).
[10] give([Igave|X],X).
[11] arrive([Iarrives|X],X).
[12] arrive([Iarrived|X],X).
[13] np([mary|X],X,mary).
[14] np([john|X],X,john).
[15] np([apples|X],X,apples).
```

The right-hand side of the first vp clause [02] cannot be executed with the reversed order of literals, since Csem, being an essential argument in np is unbound until vp is executed. However, if we do not reverse the order of vp and np literals in [02], then an infinite descend will result during generation.¹⁵ This problem could be remedied (or so it seems) by swapping the order of the vp clauses in the pair [02], [03], and allowing for the non-recursive rule to be attempted first. Specifically, we replace the two vp clauses by the following clauses (in this order):

```
[02'] vp(V1,V2,Args,Vsem) :- v(V1,V2,Args,Vsem),!.
[03'] vp(V1,V3,Args,Vsem) :- vp(V1,V2,[Csem|Args],Vsem),np(V2,V3,Csem).
```

These clauses will not be further changed by INVERSE.

However, this simple replacement will not help the generator to avoid an infinite descend when the form binding Sem does not correspond to any sentence recognizable by the parser. Instead of just swapping the clauses, we try to achieve an executable order of literals in [02] by first testing whether the stop condition, that is, clause [03], is ever likely to be used, and fail if it is not. In order to achieve this, we introduce a new non-terminals vp1, and replace clauses [02] and [03] by [021], [022], and [023] as shown below:

```
[021] vp(V1,V3,Args,Sem) :- v(V1,V2,Ar1,Sem),vp1(V2,V3,Ar1,Args).
[022] vp1(V1,V1,Args,Args) :- !.
[023] vp1(V1,V3,Ar1,Args) :- vp1(V1,V2,Ar1,[Csem|Args]),np(V2,V3,Csem).
```

In this way, the recursive rule consisting of clauses [022] and [023] cannot be used unless the semantic form Sem given as input for generation is unifiable with the semantic form of a literal which guarantees a controlled exit from the recursion, in this case v. Thus, if the recursive rule is well-formed we are guaranteed to use [022] at some point, perhaps after a few turns of [023]. Note that we let the recursion to advance one step at a time, checking if the desired bindings for other variables have been reached, that is, if Args is now unifiable with Ar1. The reader should note that the problem is not merely in removing left recursion in [03']. Indeed, doing so in a standard fashion would not do any good because as a result we would prepose the order of literals np and vp in [03']. This, however, would be subsequently reversed by INVERSE, because Csem, the essential argument in np, is not bound before vp is executed.

There is a flaw in the argument given above, however. To see it, we consider a simplified version of the "normalized" code given above.

¹⁴ A similar example, though on the meta-grammar level, is given in Shieber et al. (1989) and it is used there as an argument for introducing a mixed top-down/bottom-up evaluation strategy in generation. This becomes unnecessary, however, once the grammar is normalized or INVERSE modified to deal inter-clausal reordering.

¹⁵ Note that this program fragment is built according to quite a different design than those obtained from the string grammar. Here, both Args and Vsem are the arguments carrying the translation, or semantics, of vp non-terminal: Vsem is the template that will be filled at the lexicon level, while Args is used to collect arguments to fill this template.

```

vp(Ar1,Sem) :- v(Ar1,Sem),vp1(Ar1,Args).
vp1(Ar1,Args) :- !.
vp1(Ar1,Args) :- vp1(Ar1,f(Args)),np.

```

The reader may note that this code will work only if there exists a specific relationship between the values of variables Ar1 and Args, that is either (a) $Args = Ar1$, or (b) $Ar1 = f^n(Args)$. There is nothing in the code above to guarantee that this relationship holds. Therefore, the "normalized" code is still ill-formed, just like the original. This outcome is confirmed by the strong MSEAS procedure which fails to find any MSEA in the "normalized" code as well. What we need here is a transformation that would produce the code that would survive step (3) in strong MSEAS procedure. The general pattern is as follows:

```

vp(Ar1,Sem) :- v(Ar1,Sem),vp1(Ar1,Args).
vp1(f(Ar1),Args) :- vp1(Ar1,Args),np.
vp1(Args,Args).

```

This time we can reduce Ar1 to Args, if $Ar1 = f^n(Args)$, $n \geq 0$. If this is not the case, we stop with failure. Note that now {Ar1} is a MSEA in vp1 following the strong MSEAS procedure. If we add string variables V1, V2, etc., then we obtain also another MSEA: {V1,Ar1}. Thus, finally the code is well-formed, and reversible at that. Note also that in the end we did not have to swap the order of clauses defining vp1. This modified normalization process transforms the original program, consisting of clauses [01] to [15], into the new program, as shown below.

```

[01] sent(V1,V3,Sem) :- np(V1,V2,Ssem),vp(V2,V3,[Ssem],Sem).
[02] vp(V1,V3,Args,Sem) :- v(V1,V2,Ar1,Sem),vp1(V2,V3,Ar1,Args).
[03] vp1(V1,V3,[Csem|Ar1],Args) :-
    vp1(V1,V2,Ar1,Args),np(V2,V3,Csem).
[04] vp1(V1,V1,Args,Args).
[05] v(V1,V2,[Iobj,Obj,Subj],give(Subj,Obj,Iobj)) :- give(V1,V2).
[06] v(V1,V2,[Obj,Subj],love(Subj,Obj)) :- love(V1,V2).
[07] v(V1,V2,[Subj],arrive(Subj)) :- arrive(V1,V2).
[08] love([Iloves|X],X).
[09] love([Iloved|X],X).
[10] give([Igives|X],X).
[11] give([Igave|X],X).
[12] arrive([Iarrives|X],X).
[13] arrive([Iarrived|X],X).
[14] np([mary|X],X,mary).
[15] np([john|X],X,john).
[16] np([apples|X],X,apples).

```

After normalization, the parser program can be successfully inverted. The only change to be done by INVERSE is to reverse the order of literals in clause [01]. An obvious question to be asked at this point is how general the above method is, and what other rules of normalization might be required before the parser is fully invertible. I try to answer only the first part of this question here. The normalization rule described above seems quite general in that it does not require manipulating of literal's internal arguments, and the changes made to the grammar have always a local character. The net effect of this transformation is a code that can be effectively evaluated by a top-down interpreter, even though the original code couldn't. The code normalization stage saves the the runtime overhead that would unescapably result if we had a more elaborate interpreter, such as, for example the mixed top-down/bottom-up method proposed by Shieber et al. (1989).

In general the normalization rule discussed above can be described by the following pattern, where Sem is an essential argument in x, {Z,Sem} is a MSEA in x', and all other arguments are omitted.

<pre> x(Args,Sem) :- x(f(Args),Sem),y. x(Args,Sem) :- z(Args,Sem). z(Args,Sem) :- w. </pre>	====>	<pre> x(Args,Sem) :- z(Z,Sem),x'(Z,Args). x'(f(Z),Args) :- x'(Z,Args),y. x'(Args,Args). z(Args,Sem) :- w. </pre>
---	-------	--

There is one interesting special case of this scheme. When literal z is empty, that is $x(Args,Sem)$ in the second clause rewrites into an empty string, then we actually need to introduce one. Without it we cannot assign the initial binding to variable Z, which must be done since Z is an essential argument.

Finally, we need to decide whether the transformation described above belongs to the normalizer or perhaps it should be somehow integrated into the inversion process. This time we do not need to know beforehand if the original code is reversible, but we need to know that it is ill-formed according to the strong MSEAS procedure. This suggests that the transformation is best placed as a subprocess called from MSEAS whenever its strong version fails to find a MSEA for a clause.

4.3. Enhancing the efficiency of the generator

The method of reversing a Prolog parser described in section 3 does not yet guarantee that the obtained generator will be the optimal one. It appears, however, that if the clauses in the parser conform to certain requirements of form, a very efficient generator can be obtained. This section describes one of several types of transformations that can be used in order to put the parser into the desired form. These transformations are intended to be used before the reversal process described in the previous section is applied. The reason is that some of the transformations described below may eliminate the need for reversing of entire segments of the parser.

4.3.1. Production splitting

Normalization of the parser code prior to inversion can lead to a substantial improvement in performance of the generator. One such situation arises when some of the non-terminals on the right-hand side of some productions may, in some cases, contribute null components to the translation of the head non-terminal on the lhs. Moreover, the "semantics" of the non-terminal on the left-hand side contains no trace of the empty string production being used. In other words, the null "translation" of the non-terminal on the rhs is "spliced" into the translation of the lhs leaving no trace.

In such, and similar, situations the efficiency of the generator can be enhanced if the parser clauses are appropriately "prepared" before the reversal process starts. The transformation we want to suggest here is applicable when the "semantics" of some of the non-terminals on the right-hand side of a production contribute either uniquely identifiable components to the "semantics" of the non-terminal on the left, or else they contribute null components, in which case their presence is not recorded in the "semantics" of the left-hand side non-terminal at all. As a specific example consider the following clause implementing the string of left-modifiers to a noun (# compresses its first list argument removing all null components and deposits the result in P):

$$\text{In}(L1, L5, \text{Num}, P) :- \text{tpos}(L1, L2, \text{Num}, P1), \text{qpos}(L2, L3, \text{Num}, P2), \text{apos}(L3, L4, P3), \\ \text{npos}(L4, L5, P4), \#([P1, P2, P3, P4], P).$$

Each of tpos, qpos, apos and npos translates either into a unique component of P, such as [tpos, every] or [apos, tall], or into a nil. The procedure # would then combine lists P1 to P4, eliminating any empty list from among them. This clause belongs to the parser, and needs to be modified for the use in a generator. Our inversion algorithm requires fronting of the # literal, but one may note that this would not do much good. Given P, the generator would, in the worst case, attempt to examine all possible decompositions of the list structure in P into a four-element list. In general, the number $D(k, n)$ of different ways to decompose the list of n elements into k sublist preserving the original ordering of elements is defined by the following recursive equation:

$$D(1, n) = 1; \\ D(k, n) = \sum_{i=0}^n D(k-1, n-i) \text{ for } k \geq 2.$$

For example, $D(4, 4)=35$, $D(4, 5)=56$, $D(4, 6)=84$, $D(4, 7)=120$, etc. This is an unnecessary waste of time since we know, from the fact that components of P are uniquely identifiable, which non-terminals on the right-hand side of the clause contributed them. A better approach is to normalize the clauses so that they are better suited for the needs of generation. We may note that although the above clause works fine in a parser, it is nearly completely useless for a generator. We replace this production and all the productions rewriting tpos, qpos, apos and npos by a group of new productions that split the above clause into $2k$ new clauses where k is the number of non-terminals on the right rewriting to nil. All the clauses implementing $xpos \rightarrow \epsilon$ are eliminated, and the semantics of $xpos \rightarrow \text{const}$ is adjusted.

In general we may propose the following transformation. Given the fragment of the grammar (string variables omitted):

$$n_0(P_0) :- s_1(P_1), s_2(P_2), \dots, s_k(P_k), \#([P_1, P_2, \dots, P_k], P_0). \\ s_1([]). \\ s_2([]).$$

...
 $s_k([])$.

we replace it by the following $2*k$ productions (where $n_i, i>0$, are new non-terminals):

$n_0([P_1|P_2]) :- s_1(P_1), n_1(P_2).$
 $n_0(P) :- n_1(P).$
 $n_1([P_1|P_2]) :- s_2(P_1), n_2(P_2).$
 $n_1(P) :- n_2(P).$

 $n_{k-1}([P]) :- s_k(P).$
 $n_{k-1}([]).$

For example, the clause defining literal ln above is replaced by the following new code:

$ln3(L1,L2,[P1]) :-$
 $npos(L1,L2,P1).$
 $ln3(L1,L1,[]).$
 $ln2(L1,L3,[P1|P2]) :-$
 $ln3(L2,L3,P2),$
 $apos(L1,L2,P1).$
 $ln2(L1,L2,P2) :-$
 $ln3(L1,L2,P2).$
 $ln1(L1,L3,[P1|P2]) :-$
 $ln2(L2,L3,P2),$
 $qpos(L1,L2,P1).$
 $ln1(L1,L2,P2) :-$
 $ln2(L1,L2,P2).$
 $ln(L1,L3,[P1|P2]) :-$
 $ln1(L2,L3,P2),$
 $tpos(L1,L2,P1).$
 $ln(L1,L2,P2) :-$
 $ln1(L1,L2,P2).$

A slight variation of the above rule can be proposed in situations when some of the s_i literals do not rewrite to empty strings. Thus, given the following fragment

$n_0(P_0) :- s_1(P_1), s_2(P_2), ..., r(Q), ..., s_k(P_k), \#([P_1, P_2, ..., Q, ..., P_k], P_0).$
 $s_1([]).$
 $s_2([]).$
 ...
 $s_k([]).$

where $r(Q)$ does not rewrite to an empty string, we replace it by the following $2*k+1$ new productions:

$n_0([P_1|P_2]) :- s_1(P_1), n_1(P_2).$
 $n_0(P) :- n_1(P).$
 $n_1([P_1|P_2]) :- s_2(P_1), n_2(P_2).$
 $n_1(P) :- n_2(P).$

 $n_r([P_1|P_2]) :- r(P_1), n_{r+1}(P_2).$

 $n_{k-1}([P]) :- s_k(P).$
 $n_{k-1}([]).$

As an example consider the following clause defining the top-level structure of a "regular" noun phrase, where both ln and mr can rewrite into an empty string, but $nvar$ must always be non-empty.

$nstgo(N1,N4,[nplP]) :-$

```

ln(N1,N2,P1),
nvar(N2,N3,P2),
m(N3,N4,P3),
@(P1,[[n,P2]IP3],P).

```

This production is normalized using the modification of the splitting rule given above. The resulting program is given below.

```

nstgo2(N1,N2,P1) :-
    m(N1,N2,P1).
nstgo2(N1,N1,[]).
nstgo1(N1,N3,[[n,P2]IP3]) :-
    nstgo2(N2,N3,P3),
    nvar(N1,N2,P2).
nstgo(N1,N2,[npIP2]) :-
    nstgo1(N1,N2,P2).
nstgo(N1,N3,[npIP]) :-
    @(P1,P2,P),
    nstgo1(N2,N3,P2),
    ln(N1,N2,P1).

```

This last example shows that the splitting rule (or its variations) is applicable whenever any of the non-terminals on the right-hand side of a production rewrites to nil, even though there may be no explicit production $s_i([])$, but we may have only $s_i \rightarrow \text{nil}$. It should be stressed here that we must be careful in removing all the empty-string productions, because if we leave any around the generator will produce duplicate outputs. For example, nstgo2 above rewrites to m which is defined by the following clauses:

```

m(R1,R3,Num,[P1IP2]) :-
    mval(R1,R2,Num,P1),
    m(R2,R3,Num,P2).
m(R,R,Num,[]).

```

Thus, m can rewrite into an empty string in the second of these clauses, but we cannot simply remove it, because it also serves as a stop condition to the first recursive clause. Instead, we modify the above code as follows, which is actually in line with the transformation described above:

```

m(R1,R3,Num,[P1IP2]) :-
    mval(R1,R2,Num,P1),
    m1(R2,R3,Num,P2).
m1(R1,R3,Num,[P1IP2]) :-
    mval(R1,R2,Num,P1),
    m1(R2,R3,Num,P2).
m1(R1,R1,Num,[]).

```

4.3.2. Path compression and path merging

In some situations a special path compression algorithm may transform some of the non-local semantics chain rules into local semantics non-chain rules. If we could eliminate all non-local semantics rules, then the obtained program would actually become bi-directional, although its efficiency might have suffered due to the need for repeated re-parsing/re-generating of the same substrings. The efficiency problem may be sometimes averted if path merging is performed at the same time, the process that would collapse the paths generating from the same structure. The net effect of the combined transformations is that of "lifting" certain translation structures, associated with "semantics" arguments of literals, from non-terminals sitting lower in the grammar tree to those sitting higher up the tree. This means that some paths will be split, and some decisions will be made earlier in generation. The efficiency problem can be controlled if these early decisions are deterministic, that is, there will be no need to retrace the same steps along another path. Another problem that must be taken into account is, sometimes substantial, growth in the size of the grammar. It would be certainly undesirable if the size of the grammar, counted as a number of different productions, was directly dependent upon the size of, say, verb lexicon. Thus, the "lifting transformation" should only be used for a localized "tuning" of the parser code before its inversion.

As an example consider the following fragment of a parser:

```

assertion(A1,A41,WHQ,P) :- [1]
    sa(A1,A11,PA1),
    subject(A11,A2,WHQ,Num,P1),
    sa(A2,A21,PA2),
    verb(A21,A3,Sts,Num,P2),
    sas(A3,A31,Sts,PA3),
    object(A31,A4,O,Sts,[Num,P1],P2,PSA,P),
    sao(A4,A41,O,PA4),
    #([PA1,PA2,PA3,PA4],PSA).
object(O1,O2,tovo,Sts,P1,P2,PSA,[P2,[subject,P1],[object,P3]]PSA) :-
    tovo-verb(Sts),
    tovo(O1,O2,P1,P3).
object(O1,O2,nstgo,Sts,P1,P2,PSA,[P2,[subject,P1],[object,P3]]PSA) :-
    nstgo-verb(Sts),
    nstgo(O1,O2,Num,P3).

```

The chain rule for assertion can be replaced by two non-chain local-semantics rules eliminating the non-terminal object and compressing the three clauses into two, by expanding *object* literal in the *assertion* clause and replacing it by the right-hand side of appropriately instantiated *object* clause, as shown below:

```

assertion(A1,A41,WHQ,[P2,[subject,P1],[object,P3]]PSA) :- [2]
    sa(A1,A11,PA1),
    subject(A11,A2,WHQ,Num,P1),
    sa(A2,A21,PA2),
    verb(A21,A3,Sts,Num,P2),
    sas(A3,A31,Sts,PA3),
    tovo-verb(Sts),
    tovo(O1,O2,P1,P3),
    sao(A4,A41,O,PA4),
    #([PA1,PA2,PA3,PA4],PSA).
assertion(A1,A41,WHQ,[P2,[subject,P1],[object,P3]]PSA) :-
    sa(A1,A11,PA1),
    subject(A11,A2,WHQ,Num,P1),
    sa(A2,A21,PA2),
    verb(A21,A3,Sts,Num,P2),
    sas(A3,A31,Sts,PA3),
    nstgo-verb(Sts),
    nstgo(O1,O2,Num,P3),
    sao(A4,A41,O,PA4),
    #([PA1,PA2,PA3,PA4],PSA).

```

The problem with this code is that we have no way of telling which of the two clauses should be used in any particular case of generation, because they both require the same structure of the underlying form. Moreover, if we fire the wrong one, which we do not know until reaching *tovo* or *nstgo* literals, we will have to back-up, throwing away successfully generated subject, verb, and perhaps some sentence adjuncts, only to re-generate them anew while evaluating the other clause. To correct this we merge fragments of the rhs's of both clauses, and break them back into three clauses, as shown below:

```

assertion(A1,A41,WHQ,[P2,[subject,P1],[object,P3]]PSA) :- [3]
    !,sa(A1,A11,PA1),
    subject(A11,A2,WHQ,Num,P1),
    sa(A2,A21,PA2),
    verb(A21,A3,Sts,Num,P2),
    sas(A3,A31,Sts,PA3),
    object1(A31,A4,O,Sts,P1,P2,PSA,P3),
    sao(A4,A41,O,PA4),
    #([PA1,PA2,PA3,PA4],PSA).
object1(O1,O2,tovo,Sts,P1,P2,PSA,P3) :-

```

```

    tovo-verb(Sts),
    tovo(O1,O2,P1,P3).
object1(O1,O2,nstgo,Sts,P1,P2,PSA,P3) :-
    nstgo-verb(Sts),
    nstgo(O1,O2,Num,P3).

```

As the result, we lifted the translation scheme $[P2, [subject, P1], [object, P3] | PSA]$ from object to assertion. In this way, the decision to take a specific path in generation can be made earlier in the process, and some of the non-local semantics rules can be eliminated. This, in turn, reduces the need for code inversion. After this transformation the assertion clause is nearly bi-directional, except for the list splicing primitive which, for generation, must be placed in front of all sentence adjunct literals. We use *object1* in [3] rather than *object* to prevent other object clauses, with different semantics, to be fired from the rhs of assertion. To illustrate this, we add two more clauses to [1], as shown in [4].

```

object(O1,O2,vo,Sts,P1,P2,PSA,[P2,P3|PSA]) :-                                     [4]
    vo(O1,O2,P1,P3).
vo(V1,V41,P1,P4) :-
    vb(V1,V2,Sts,inf,P2),
    sas(V2,V3,Sts,PA1),
    object(V3,V4,O,Sts,P1,P2,PSA,P4),
    sao(V4,V41,O,PA2),
    @(PA1,PA2,PSA).

```

When the "lifting" transformation is applied to [1] and [4] taken together, four new clause are added to [3], as shown in [5] below. The reader may note the new non-terminal *object2* which is called only if the pattern $[P2, P3 | PSA]$ is recognized at the assertion level.

```

assertion(A1,A41,WHQ,[P2,P3|PSA]) :-                                           [5]
    !,sa(A1,A11,PA1),
    subject(A11,A2,WHQ,Num,P1),
    sa(A2,A21,PA2),
    verb(A21,A3,Sts,Num,P2),
    sas(A3,A31,Sts,PA3),
    object2(A31,A4,O,Sts,P1,P2,PSA,P3),
    sao(A4,A41,O,PA4),
    #([PA1,PA2,PA3,PA4],PSA).
object2(O1,O2,vo,Sts,P1,P2,PSA,P3) :-
    vo(O1,O2,P1,P3).
vo(V1,V41,P1,[P2,[subject,P1],[object,P3]|PSA]) :-
    !,vb(V1,V2,Sts,inf,P2),
    sas(V2,V3,Sts,PA1),
    object1(V3,V4,O,Sts,P1,P2,PSA,P3),
    sao(V4,V41,O,PA2),
    @(PA1,PA2,PSA).
vo(V1,V41,P1,[P2,P3|PSA]) :-
    !,vb(V1,V2,Sts,inf,P2),
    sas(V2,V3,Sts,PA1),
    object2(V3,V4,O,Sts,P1,P2,PSA,P3),
    sao(V4,V41,O,PA2),
    @(PA1,PA2,PSA).

```

The reader should note that the lifting transformation must also take care of the arguments to the new non-terminals (here, *object1* and *object2*). Since these will be derived from the arguments to the old non-terminal (here, *object*), the general rule is as follows:

- (1) Initially, the new non-terminal, say $X1$, inherits all the arguments of the old non-terminal, say X , which are not explicitly lifted. (In our last example, $[P2, P3 | PSA]$ is explicitly lifted, so *object2* has initially the following arguments: *object2*($O1, O2, vo, Sts, P1, P2, PSA$).)
- (2) Add, as new arguments, those free variables found in the lifted pattern which are not already arguments to $X1$. (In our example, we need to add $P3$ to *object2*.)

- (3) Remove those arguments from *X1* which contain none of the variables appearing on the right-hand side of any of the clauses defining an expansion for *X1*. (If we assume that the only expansion for *object2* is *vo(O1,O2,P1,P3)*, then *P2* can be dropped as an argument to *object2*.)

Finally, we may note that the lifting transformation is also applicable in cases when the patterns are put together by dedicated literals, such as *combine*, *@* or *#*, rather than by ready-made terms. In such cases we lift the pattern-forming literal, removing it from its present location and placing it in front of the right-hand side of the clause to which the lifting is made. All other aspects of the transformation remain unchanged. For example, if we had

```
object(O1,O2,vo,Sts,P1,P2,PSA,P) :-
    vo(O1,O2,P1,P3),
    combine([P2,P3,PSA],P).
```

then the resulting *assertion* clause would be:

```
assertion(A1,A41,WHQ,P) :-
    combine([P2,P3,PSA],P),
    sa(A1,A11,PA1),
    subject(A11,A2,WHQ,Num,P1),
    sa(A2,A21,PA2),
    verb(A21,A3,Sts,Num,P2),
    sas(A3,A31,Sts,PA3),
    object2(A31,A4,O,Sts,P1,P2,PSA,P3),
    sao(A4,A41,O,PA4),
    #([PA1,PA2,PA3,PA4],PSA).
```

Last, but not the least, it must be stressed here that in order to benefit from the "lifting" transformation the generator must be able to make deterministic choices among the paths with lifted semantics, or else it will be running through costly back-ups.¹⁶ This, however, may not always be possible. Consider, for example, the following clause.

```
object(O1,O2,objectbe,Sts,P1,P2,PSA,P3) :-
    objectbe-verb(Sts),!,
    objectbe(O1,O2,P1,P2,PSA,P3).
```

Here lifting of variable *P3* from *object* to *assertion* does not mark a unique generation path, because anything can be unified against an uninstantiated variable. In such a case the lifting transformation makes little sense and we are better off without it. Before giving up, however, we should consider a possibility that the unique semantic pattern we are looking for can be lifted from literals located even further below in the grammar tree. This is in fact the case with the *object* clause above, because we can lift patterns associated with clauses defining *objectbe*. In other situations, when two patterns are not sufficiently different, they may be made more specific by lifting fragments of translation structure from the literals further below in order to replace some of the free variables in these patterns. On the other hand, we do not want to go too deep into the grammar tree to find our unique patterns. On the one extreme, we might have to go as far as the lexicon, which would have the unfortunate effect of making the size of the grammar proportional to the size of the lexicon. The reader may note that the following grammar is not a good choice for the lifting transformation.

```
sent(Sem) :- np, vp(Sem).
vp(Sem) :- vp(Sem), np.
vp(gives(X,Y,Z)).
vp(likes(X,Y)).
vp(walks(X)).
```

The basic version of the "lifting" transformation can be implemented using a fairly straightforward algorithm. A more interesting version that would take into account the problems discussed here is still under investigation.

¹⁶ Excessive backtracking can be prevented by using the cut, as shown in the examples above. However, the cut also eliminates non-determinism, and thus reduces the power of a non-deterministic grammar.

5. Conclusions

In this paper we presented an algorithm for automatic inversion of a unification parser for natural language into an efficient unification generator. The major characteristics of this algorithm include: (1) integration of context-free grammar rules with context-sensitive rules such as morphological agreement and "semantic" translations, within a single inversion; (2) full automatization of the inversion process; and (3) maximum efficiency of the generator obtained as the result of inversion. In fact, the inverted parser generator behaves as if it was "parsing" the output of the non-inverted parser. This is in contrast with the methods where the inverted parser generator operates by making a number of guesses (as to the final appearance of the surface string) which are subsequently verified or rejected by parsing each guessed string or word and comparing so obtained structure with what is known to the generator.

This paper is not an attempt to define a general algorithm for reversing programs or functions, such as those discussed by McCarthy (1956) or Dijkstra (1983), among others. The method relies, to some extent, on the specific properties of a unification-based parser for English, and its top-down, left-to-right character. One of the problems remaining to be worked out is a precise definition of what type a grammar/parser can be most profitably inverted using the method presented here. The reader may also have noted that we did not discuss what other problems may need to be solved during the normalization phase, besides the few cases discussed in section 4.

6. Acknowledgements

Ralph Grishman, Ping Peng, and other members of the NYU's Natural Language Discussion Group contributed their comments and criticism to the earlier versions of this paper.

7. References

- Dijkstra, E. W. (1983). *Program inversion*. EWD671, Springer, pp. 351-354.
- Dymetman, M., P. Isabelle (1988). "Reversible Logic Grammars for Machine Translation." Proc. CMU MT Conference.
- McCarthy, J. (1956). "The Inversion of Functions Defined by Turing Machines." In C.E. Shannon, J. McCarthy (eds.), *Automata Studies*, Princeton University Press.
- N. Sager (1981). *Natural Language Information Processing*. Addison-Wesley.
- S. M. Shieber, G. van Noord, R. C. Moore, F. C. N. Pereira, *A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms*. Proceedings of the 27th Meeting of the ACL, Vancouver, B.C. (1989), pp. 7-17.
- Shoham, Y., D. V. McDermott (1984). "Directed Relations and Inversion of Prolog Programs." Proc. of the Int. Conference of Fifth Generation Computer Systems.
- Strzalkowski, T. (1989). *An algorithm for inverting a unification grammar into an efficient unification generator*. To appear in *Applied Mathematics Letters*.

Appendices

Appendix A list the Prolog code of an actual unification parser for a fragment of English. Appendix B gives the code of the generator obtained by inverting this parser using the algorithm presented in this paper. Finally, Appendix C lists a fragment of the lexicon used by both the parser and the generator.

Appendix A: A Prolog parser for a fragment of English

```

@([X|L1],L2,[X|L3]) :- @(L1,L2,L3).
@([],L,L).
#([X|L1],L2) :- #(L1,L3),@(X,L3,L2),!.
#([],[]).
mem(X,[X|L]) :- !.
mem(X,[Y|L]) :- mem(X,L).
parse(S,P) :- sentence(S,[],P).
sentence(S1,S2,[assert,P]) :- assertion(S1,S2,asn,nogap,P).
sentence(S1,S2,P) :- question(S1,S2,P).
sentence(S1,S2,P) :- imperative(S1,S2,P).
assertion(A1,A41,WHQ,Gap,P) :-
    sa(A1,A11,PA1),
    subject(A11,A2,WHQ,Num,P1),
    sa(A2,A21,PA2),
    verb(A21,A3,Sts1,Num,P2),
    sas(A3,A31,Sts1,Sts2,PA3),
    object(A31,A4,O,Sts2,Gap,assert,[Num,P1],P2,PSA,P),
    sao(A4,A41,O,PA4),
    #([PA1,PA2,PA3,PA4],PSA).
question(Q1,Q2,[askif,P]) :- nesnoq(Q1,Q2,ynq,P).
question(Q1,Q2,P) :- whqn(Q1,Q2,P).
question(Q1,Q2,P) :- whq(Q1,Q2,P).
imperative(I1,I3,[command,P]) :-
    sa(I1,I2,PSA),
    vo(I2,I3,Num,you,P2),
    @(P2,PSA,P).
yesnoq(A1,A41,WHQ,P) :-
    auxverb(A1,A2,Sts,Num,P2),
    subject(A2,A3,WHQ,Num,P1),
    sa(A3,A31,PA1),
    object(A31,A4,O,Sts,nogap,yesno,[Num,P1],P2,PSA,P),
    sao(A4,A41,O,PA2),
    @(PA1,PA2,PSA).
whqn(W1,W4,[askwh,P2,P3|PSA]) :-
    sa(W1,W2,PSA),
    intro(W2,W3,P2),
    restq(W3,W4,P3).
restq(R1,R2,P) :- yesnoq(R1,R2,whq,P).
restq(R1,R2,P) :- assertion(R1,R2,whq,nogap,P).
restq(R1,R1,[]).
intro(I1,I2,[np,P]) :- whword(I1,I2,P).
whq(W2,W4,[P2,P3]) :-
    wher2(W2,W3,P2),
    yesnoq(W3,W4,whq,P3).
wher2(W1,W2,P) :- whques(W1,W2,P).
wher2(W1,W1,[]).
whnqn(W1,W3,[askwh,P1,P2]) :-
    whn(W1,W2,P1),
    restq(W2,W3,P2).
whn(W1,W2,P) :- nstgo(W1,W2,Num,nogap,P).
object(O1,O2,vo,Sts,Gap,yesno,P1,P2,PSA,[P2,P3|PSA]) :-
    vo(O1,O2,Gap,Num,P1,P3).
object(O1,O2,tovo,Sts,Gap,N,P1,P2,PSA,[P2,[subject,P1],[object,P3]|PSA]) :-
    tovo-verb(Sts),
    tovo(O1,O2,Gap,P1,P3).

```



```

object(O1,O2,nstgo,Sts,Gap,N,P1,P2,PSA,[P2,[subject,P1],[object,P3]|PSA]) :-
    nstgo-verb(Sts),
    nstgo(O1,O2,Num,Gap,P3).
object(O1,O2,null,Sts,nogap,N,P1,P2,PSA,[P2,[subject,P1]|PSA]) :-
    nullobj-verb(Sts),
    nullobj(O1,O2).
object(O1,O2,thats,Sts,Gap,N,P1,P2,PSA,[P2,[subject,P1],[object,P3]|PSA]) :-
    thats(O1,O2,Gap,P3).
object(O1,O2,pn,Sts,Gap,N,P1,P2,PSA,[P2,[subject,P1],[object,P3]|PSA]) :-
    pn-verb(Sts,PP),
    pn(O1,O2,Gap,PP,P3).
object(O1,O2,objectbe,Sts,Gap,N,P1,P2,PSA,P3) :-
    objectbe-verb(Sts),!,
    objectbe(O1,O2,Gap,P1,P2,PSA,P3).
objectbe(O1,O2,nogap,P1,P2,PSA,P) :-
    objbe(O1,O2,P1,P3),
    @(P3,PSA,P).
objectbe(O1,O2,Gap,P1,P2,PSA,P) :-
    venpass(O1,O2,Gap,P1,P3),
    @([P2,P3],PSA,P).
objectbe(O1,O2,Gap,P1,P2,PSA,P) :-
    vingo(O1,O2,Gap,P1,P3),
    @([P2,P3],PSA,P).
nullobj(N,N).
objbe(O1,O2,P1,[P3,[subject,P1]]) :-
    adj(O1,O2,P3).
objbe(O1,O2,P1,[be,[subject,P1],[object,P3]]) :-
    nstgo(O1,O2,Num,nogap,P3).
objbe(O1,O2,P1,[be,[subject,P1],P3]) :-
    pn(O1,O2,nogap,PP,P3).
passobj(S1,S2,pn,P1,P2,[object,P1],P3) :-
    pn(S1,S2,nogap,PP,P3).
passobj(S1,S2,objbe,P1,P2,[object,P3]) :-
    objbe(S1,S2,P1,P3).
passobj(S1,S1,null,P1,P2,[object,P1]).
tovo(T1,T41,Gap,P1,P) :-
    to(T1,T2),
    verb(T2,T3,Sts1,inf,P2),
    sas(T3,T31,Sts1,Sts2,PA1),
    object(T31,T4,O,Sts2,Gap,tovo,P1,P2,PSA,P),
    sao(T4,T41,O,PA2),
    @(PA1,PA2,PSA).
nstgo(N,N,Num,var,[np-gap,var]).
nstgo(N1,N4,Num,nogap,[np|P]) :-
    ln(N1,N2,Num,P1),
    nvar(N2,N3,Num,P2),
    rn(N3,N4,Num,P3),
    @(P1,[n,P2]|P3),P).
ln(L1,L5,Num,P) :-
    tpos(L1,L2,Num,P1),
    qpos(L2,L3,Num,P2),
    apos(L3,L4,P3),
    npos(L4,L5,P4),
    #([P1,P2,P3,P4],P).
tpos(T1,T2,Num,[t-pos,P]) :-
    det(T1,T2,Num,P).

```

```

tpos(T1,T2,Num,[[t-pos,P]]) :-
    whln(T1,T2,P).
tpos(T1,T2,Num,P) :-
    howqstg(T1,T2,P).
tpos(T,T,Num,[]).
whln([what|W],W,what).
whln([which|W],W,which).
howqstg([how,many|H],H,[howmany]).
howqstg([how,much|H],H,[howmuch]).
qpos(Q1,Q2,Num,[[count,P]]) :-
    count(Q1,Q2,Num,P).
qpos(Q,Q,Num,[]).
apos(A1,A2,[[adj,P]]) :-
    adj(A1,A2,P).
apos(A1,A2,[[a-pos-ving,P]]) :-
    ving(A1,A2,S,P).
apos(A1,A2,[[qn-pos,P]]) :-
    qn(A1,A2,P).
apos(A,A,[]).
npos(N1,N3,[[n-pos,[np,P]]]) :-
    lcdn(N1,N2,P1),
    noun(N2,N3,Num,P2),
    @(P2,P1,P).
npos(N,N,[]).
lcdn(L1,L2,[[adj,P]]) :-
    adj(L1,L2,P).
lcdn(L,L,[]).
qn(Q1,Q3,[np,[noun,P2],[count,P1]]) :-
    count(Q1,Q2,Num,P1),
    noun(Q2,Q3,Num,P2).
nvar(N1,N2,Num,P) :- noun(N1,N2,Num,P).
nvar(N1,N2,Num,P) :- pro(N1,N2,Num,P).
rn(R1,R3,Num,[P1|P2]) :-
    rnval(R1,R2,Num,P1),
    rn(R2,R3,Num,P2).
rn(R,R,Num,[]).
rnval(R1,R2,Num,P) :- pn(R1,R2,nogap,PP,P).
rnval(R1,R2,Num,[rn-wh|P]) :-
    vingo(R1,R2,nogap,var,P).
rnval(R1,R2,Num,[rn-wh|P]) :-
    venpass(R1,R2,nogap,var,P).
rnval(R1,R3,Num,[rn-wh,P]) :-
    wh-that(R1,R2),
    vo(R2,R3,nogap,Num,var,P).
rnval(R1,R3,Num,[rn-wh,P]) :-
    wh-that(R1,R2),
    assertion(R2,R3,rn-wh,var,P).
pn(N1,N3,Gap,PP,[P1,P2]) :-
    prep(N1,N2,P1),
    mem(P1,PP),
    nstgo(N2,N3,Num,Gap,P2).
subject(S1,S2,WHQ,Num,P) :- nstgo(S1,S2,Num,nogap,P).
subject(S1,S1,whq,Num,var).
passagent(A1,A3,Gap,P) :-
    by(A1,A2,P1),
    nstgo(A2,A3,Num,Gap,P).

```

```

thats(T1,T3,Gap,P) :-
    that(T1,T2),
    assertion(T2,T3,thats,Gap,P).
vo(V1,V41,Gap,Num,P1,P4) :-
    vb(V1,V2,Sts1,Num,P2),
    sas(V2,V3,Sts1,Sts2,PA1),
    object(V3,V4,O,Sts2,Gap,vo,P1,P2,PSA,P4),
    sao(V4,V41,O,PA2),
    @(PA1,PA2,PSA).
vingo(V1,V41,Gap,P1,P4) :-
    ving(V1,V2,Sts1,P2),
    sas(V2,V3,Sts1,Sts2,PA1),
    object(V3,V4,O,Sts2,Gap,vingo,P1,P2,PSA,P4),
    sao(V4,V41,O,PA2),
    @(PA1,PA2,PSA).
veno(V1,V41,Gap,P1,P4) :-
    ven(V1,V2,Sts1,P2),
    sas(V2,V3,Sts1,Sts2,PA1),
    object(V3,V4,O,Sts2,Gap,veno,P1,P2,PSA,P4),
    sao(V4,V41,O,PA2),
    @(PA1,PA2,PSA).
venpass(V1,V5,nogap,P1,P) :-
    ven(V1,V2,Sts,P2),
    sa(V2,V3,PA1),
    passobj(V3,V4,O,P1,P2,P4),
    sao(V4,V5,O,PA2),
    #([P2,[subject,anyone]|P4],PA1,PA2,P).
venpass(V1,V6,Gap,P1,P) :-
    ven(V1,V2,S,P2),
    sa(V2,V3,PA1),
    passagent(V3,V4,Gap,P4),
    passobj(V4,V5,O,P1,P2,P5),
    sao(V5,V6,O,PA2),
    #([P2,[subject,P4]|P5],PA1,PA2,P).
venpass(V1,V6,Gap,P1,P) :-
    ven(V1,V2,S,P2),
    sa(V2,V3,PA1),
    passobj(V3,V4,pn,P1,P2,P4),
    passagent(V4,V5,Gap,P5),
    sa(V5,V6,PA2),
    #([P2,[subject,P5]|P4],PA1,PA2,P).
sao(S1,S1,null,[]) :- !.
sao(S1,S2,null,P) :- !,fail.
sao(S1,S2,O,P) :- sas(S1,S2,[any],PV,P).
sa(S1,S2,P) :- sas(S1,S2,[any],PV,P).
sas(S1,S3,PV1,PV3,[P1|P2]) :-
    saval(S1,S2,PV1,PV2,P1),
    sas(S2,S3,PV2,PV3,P2).
sas(S1,S1,PV1,PV2,[]) :- unistr(PV1,PV2).
saval(S1,S2,PV,PV,[adv,P]) :- adv(S1,S2,P).
saval(S1,S2,PV1,PV2,P) :- saval1(S1,S2,PV1,[],PV2,P).
saval1(S1,S2,[tovo|PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[ntovo|PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[nstgo|PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[pnl|PN]|PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[objectbel|PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[OTHER|PV],PV1,PV2,P) :-

```

```

    !,saval1(S1,S2,PV,[OTHER|PV1],PV2,P).
saval1(S1,S2,[],[],P) :- !,fail.
saval1(S1,S2,[],PV,PV,[in-order-to,P]) :- tovo(S1,S2,nogap,pro,P).
saval1(S1,S2,[],PV,PV,P) :- pn(S1,S2,nogap,PP,P).
objectbe-verb(Sts) :- mem(objectbe,Sts).
tovo-verb(Sts) :- mem(tovo,Sts).
nstgo-verb(Sts) :- mem(nstgo,Sts).
nullobj-verb(Sts) :- mem(nullobj,Sts).
pn-verb(Sts,PP) :- mem([pn|PP],Sts).
vb(X,Y,Sts,Num,V) :- verb(X,Y,Sts,Num,V),!.
vb(X,Y,Sts,Num,V) :- ven(X,Y,Sts,V),!.
vb(X,Y,Sts,Num,V) :- ving(X,Y,Sts,V),!.
auxverb(V1,V2,Sts,Num,P) :- def(verb,P,aux,Sts),!,vrb(V1,V2,Num,P).
auxverb(V1,V2,Sts,Num,P) :- vrb(V1,V2,Num,P),def(verb,P,aux,Ost),@(X,Sts,Ost).
verb(V1,V2,Sts,Num,P) :- vrb(V1,V2,Num,P),def(verb,P,Aux,Sts),!.
verb(V1,V2,Sts,Num,P) :- vrb(V1,V2,Num,P),def(verb,P,Aux,Ost),@(X,Sts,Ost).
ven(V1,V2,Sts,P) :- vrb(V1,V2,en,P),def(verb,P,Aux,Sts),!.
ven(V1,V2,Sts,P) :- vrb(V1,V2,en,P),def(verb,P,Aux,Ost),@(X,Sts,Ost).
ving(V1,V2,Sts,P) :- vrb(V1,V2,ing,P),def(verb,P,Aux,Sts),!.
ving(V1,V2,Sts,P) :- vrb(V1,V2,ing,P),def(verb,P,Aux,Ost),@(X,Sts,Ost).
unistr(S,S) :- !.
unistr([EIS1],[EIS2]) :- unistr(S1,S2).
unistr([EIS1],S2) :- @(X,[EIY],S2),@(X,Y,S3),!,unistr(S1,S3).

```

Appendix B: An inverted-parser generator for a fragment of English

```

@([],L,L).
@([X|L1],L2,[X|L3]) :- @(L1,L2,L3).
#([X|L1],L2) :- @(X,L3,L2),#(L1,L3).
#([],[]).
mem(X,[X|L]) :- !.
mem(X,[Y|L]) :- mem(X,L).
parse(S,P) :- sentence(S,[],P).
sentence(S1,S2,[assert,P]) :- assertion(S1,S2,asn,nogap,P).
sentence(S1,S2,P) :- question(S1,S2,P).
sentence(S1,S2,P) :- imperative(S1,S2,P).
assertion(A1,A41,WHQ,Gap,P) :-
    object(A31,A4,O,Sts2,Gap,assert,[Num,P1],P2,PSA,P),
    subject(A11,A2,WHQ,Num,P1),
    verb(A21,A3,Sts1,Num,P2),
    #([PA1,PA2,PA3,PA4],PSA),
    sa(A1,A11,PA1),
    sa(A2,A21,PA2),
    sas(A3,A31,Sts1,Sts2,PA3),
    sao(A4,A41,O,PA4).
question(Q1,Q2,[askif,P]) :- yesnoq(Q1,Q2,ynq,P).
question(Q1,Q2,P) :- whqn(Q1,Q2,P).
question(Q1,Q2,P) :- whq(Q1,Q2,P).
imperative(I1,I3,[command,P]) :-
    @(P2,PSA,P),
    sa(I1,I2,PSA),
    vo(I2,I3,Num,you,P2).
yesnoq(A1,A6,WHQ,P) :-
    object(A4,A5,O,Sts,nogap,yesno,[Num,P1],P2,PSA,P),
    auxverb(A1,A2,Sts,Num,P2),
    subject(A2,A3,WHQ,Num,P1),
    @([PA1,PA2,PA3],PSA),
    sa(A3,A4,PA1),
    sao(A5,A6,O,PA2).
whqn(W1,W4,[askwh,P2,P3|PSA]) :-
    sa(W1,W2,PSA),
    intro(W2,W3,P2),
    restq(W3,W4,P3).
restq(R1,R2,P) :- yesnoq(R1,R2,whq,P).
restq(R1,R2,P) :- assertion(R1,R2,whq,P).
restq(R1,R1,[]).
intro(I1,I2,[np,P]) :- whword(I1,I2,P).
whq(W2,W4,[P2,P3]) :-
    wher2(W2,W3,P2),
    yesnoq(W3,W4,whq,P3).
wher2(W1,W1,[]).
wher2(W1,W2,P) :- whques(W1,W2,P).
whnqn(W1,W2,[askwh,P1]) :-
    whn(W1,W2,P1),!.
whnqn(W1,W3,[askwh,P1,P2]) :-
    whn(W1,W2,P1),
    restq(W2,W3,P2).
whn(W1,W2,P) :- nstgo(W1,W2,Num,nogap,P).
object(O1,O2,tovo,Sts,Gap,N,P1,P2,PSA,[P2,[subject,P1],[object,P3]|PSA]) :-
    tovo-verb(Sts),
    tovo(O1,O2,Gap,P1,P3).

```

```

object(O1,O2,nstgo,Sts,Gap,N,P1,P2,PSA,[P2,[subject,P1],[object,P3]|PSA]) :-
    nstgo-verb(Sts),
    nstgo(O1,O2,Num,Gap,P3).
object(O1,O2,null,Sts,nogap,N,P1,P2,PSA,[P2,[subject,P1]|PSA]) :-
    nullobj-verb(Sts),
    nullobj(O1,O2).
object(O1,O2,vo,Sts,Gap,yesno,P1,P2,PSA,[P2,P3|PSA]) :-
    vo(O1,O2,Gap,Num,P1,P3).
object(O1,O2,veno,Sts,Gap,N,P1,P2,PSA,[perf,[P2|P3]|PSA]) :-
    veno(O1,O2,Gap,P1,P3).
object(O1,O2,thats,Sts,Gap,N,P1,P2,PSA,[P2,[subject,P1],[object,P3]|PSA]) :-
    thats(O1,O2,Gap,P3).
object(O1,O2,pn,Sts,Gap,N,P1,P2,PSA,[P2,[subject,P1],[object,P3]|PSA]) :-
    pn-verb(Sts,PP),
    pn(O1,O2,Gap,PP,P3).
object(O1,O2,objectbe,Sts,Gap,N,P1,P2,PSA,P3) :-
    objectbe(O1,O2,Gap,P1,P2,PSA,P3),
    objectbe-verb(Sts).
objectbe(O1,O2,nogap,P1,P2,PSA,P) :-
    @(P3,PSA,P),
    objbe(O1,O2,P1,P3).
objectbe(O1,O2,Gap,P1,P2,PSA,P) :-
    @([P2,P3],PSA,P),
    venpass(O1,O2,Gap,P1,P3).
objectbe(O1,O2,Gap,P1,P2,PSA,P) :-
    @([P2,P3],PSA,P),
    vingo(O1,O2,Gap,P1,P3).
nullobj(N,N).
objbe(O1,O2,P1,[P3,[subject,P1]]) :-
    adj(O1,O2,P3).
objbe(O1,O2,P1,[be,[subject,P1],[object,P3]]) :-
    nstgo(O1,O2,Num,nogap,P3).
objbe(O1,O2,P1,[be,[subject,P1].P3]) :-
    pn(O1,O2,nogap,PP,P3).
passobj(S1,S2,pn,P1,P2,[object,P1],P3) :-
    pn(S1,S2,nogap,PP,P3).
passobj(S1,S2,objbe,P1,P2,[object,P3]) :-
    objbe(S1,S2,P1,P3).
passobj(S1,S1,null,P1,P2,[object,P1]).
tovo(T1,T41,Gap,P1,P) :-
    object(T31,T4,O,Sts2,Gap,tovo,P1,P2,PSA,P),
    to(T1,T2),
    verb(T2,T3,Sts1,inf,P2),
    @(PA1,PA2,PSA),
    sas(T3,T31,Sts1,Sts2,PA1),
    sao(T4,T41,O,PA2).
nstgo2(N1,N2,Num,P1) :-
    rn(N1,N2,Num,P1).
nstgo2(N1,N1,Num,[]).
nstgo1(N1,N3,Num,[n,P2]|P3) :-
    nvar(N1,N2,Num,P2),
    nstgo2(N2,N3,Num,P3).
nstgo(N,N,Num,var,[np-gap,var]).
nstgo(N1,N2,Num,nogap,[np|P2]) :-
    nstgo1(N1,N2,Num,P2).

```

```

nstgo(N1,N3,Num,nogap,[np|P]) :-
    @(P1,P2,P),
    ln(N1,N2,Num,P1),
    nstgo1(N2,N3,Num,P2).
ln3(L1,L2,[P1]) :-
    npos(L1,L2,P1).
ln3(L1,L1,[]).
ln2(L1,L3,[P1|P2]) :-
    apos(L1,L2,P1),
    ln3(L2,L3,P2).
ln2(L1,L2,P2) :-
    ln3(L1,L2,P2).
ln1(L1,L2,Num,[P1|P2]) :-
    qpos(L1,L2,Num,P1),
    ln2(L2,L3,P2).
ln1(L1,L2,Num,P2) :-
    ln2(L1,L2,P2).
ln(L1,L3,Num,[P1|P2]) :-
    tpos(L1,L2,Num,P1),
    ln1(L2,L3,Num,P2).
ln(L1,L2,Num,P2) :-
    ln1(L1,L2,Num,P2).
tpos(T1,T2,Num,[t-pos,P]) :-
    det(T1,T2,Num,P).
tpos(T1,T2,Num,[t-pos,P]) :-
    whln(T1,T2,P).
tpos(T1,T2,Num,P) :-
    howqstg(T1,T2,P).
whln([what|W],W,what).
whln([which|W],W,which).
howqstg([how,many|H],H,howmany).
howqstg([how,much|H],H,howmuch).
qpos(Q1,Q2,Num,[count,P]) :-
    count(Q1,Q2,Num,P).
apos(A1,A2,[adj,P]) :-
    adj(A1,A2,P).
apos(A1,A2,[a-pos-ving,P]) :-
    ving(A1,A2,Sts,P).
apos(A1,A2,[qn-pos,P]) :-
    qn(A1,A2,P).
npos(N1,N2,[n-pos,[np,[n,P2]]]) :-
    noun(N1,N2,Num,P2).
npos(N1,N3,[n-pos,[np,[n,P2],P1]]) :-
    lcdn(N1,N2,P1),
    noun(N2,N3,Num,P2).
lcdn(L1,L2,[adj,P]) :-
    adj(L1,L2,P).
qn(Q1,Q3,[np,[noun,P2],[count,P1]]) :-
    count(Q1,Q2,Num,P1),
    noun(Q2,Q3,Num,P2).
nvar(N1,N2,Num,P) :- noun(N1,N2,Num,P).
nvar(N1,N2,Num,P) :- pro(N1,N2,Num,P).
m(R1,R3,Num,[P1|P2]) :-
    mval(R1,R2,Num,P1),
    m1(R2,R3,Num,P2).

```

```

rn1(R1,R3,Num,[P1|P2]) :-
    rnval(R1,R2,Num,P1),
    rn1(R2,R3,Num,P2).
rn1(R1,R1,Num,[]).
rnval(R1,R2,Num,P) :- pn(R1,R2,nogap,PP,P).
rnval(R1,R2,Num,[rn-wh|P]) :-
    vingo(R1,R2,nogap,var,P).
rnval(R1,R2,Num,[rn-wh|P]) :-
    venpass(R1,R2,nogap,var,P).
rnval(R1,R3,Num,[rn-wh,P]) :-
    wh-that(R1,R2),
    vo(R2,R3,nogap,Num,var,P).
rnval(R1,R3,Num,[rn-wh,P]) :-
    wh-that(R1,R2),
    assertion(R2,R3,rn-wh,var,P).
pn(N1,N3,Gap,PP,[P1,P2]) :-
    prep(N1,N2,P1),
    mem(P1,PP),
    nstgo(N2,N3,Num,Gap,P2).
subject(S1,S2,WHQ,Num,P) :- nstgo(S1,S2,Num,nogap,P).
subject(S1,S1,whq,Num,var).
passagent(A1,A3,Gap,P) :-
    by(A1,A2,P1),
    nstgo(A2,A3,Num,Gap,P).
thats(T1,T3,Gap,P) :-
    that(T1,T2),
    assertion(T2,T3,thats,Gap,P).
vo(V1,V41,Gap,Num,P1,P4) :-
    object(V3,V4,O,Sts2,Gap,vo,P1,P2,PSA,P4),
    vb(V1,V2,Sts1,Num,P2),
    @(PA1,PA2,PSA),
    sas(V2,V3,Sts1,Sts2,PA1),
    sao(V4,V41,O,PA2).
vingo(V1,V41,Gap,P1,P4) :-
    object(V3,V4,O,Sts2,Gap,vingo,P1,P2,PSA,P4),
    ving(V1,V2,Sts1,P2),
    @(PA1,PA2,PSA),
    sas(V2,V3,Sts1,Sts2,PA1),
    sao(V4,V41,O,PA2).
veno(V1,V41,Gap,P1,P4) :-
    object(V3,V4,O,Sts2,Gap,veno,P1,P2,PSA,P4),
    ven(V1,V2,Sts1,P2),
    @(PA1,PA2,PSA),
    sas(V2,V3,Sts1,Sts2,PA1),
    sao(V4,V41,O,PA2).
venpass(V1,V5,Gap,P1,P) :-
    @([P2,[subject,anyone]|P4],PSA,P),
    ven(V1,V2,Sts,P2),
    passobj(V3,V4,O,P1,P2,P4),
    @(PA1,PA2,PSA),
    sa(V2,V3,PA1),
    sao(V4,V5,O,PA2).

```



```

venpass(V1,V6,Gap,P1,P) :-
    @([P2,[subject,P4]!P5],PSA,P),
    ven(V1,V2,S,P2),
    passobj(V4,V5,O,P1,P2,P5),
    passagent(V3,V4,Gap,P4),
    @(PA1,PA2,PSA),
    sa(V2,V3,PA1),
    sao(V5,V6,O,PA2).
venpass(V1,V6,Gap,P1,P) :-
    @([P2,[subject,P5]!P4],PSA,P),
    ven(V1,V2,S,P2),
    passobj(V3,V4,pn,P1,P2,P4),
    passagent(V4,V5,Gap,P5),
    @(PA1,PA2,PSA),
    sa(V2,V3,PA1),
    sa(V5,V6,PA2).
sao(S1,S1,null,[]) :- !.
sao(S1,S2,null,P) :- !,fail.
sao(S1,S2,O,P) :- sas(S1,S2,[any],PV,P).
sa(S1,S2,P) :- sas(S1,S2,[any],PV,P).
sas(S1,S3,PV1,PV3,[P1!P2]) :-
    saval(S1,S2,PV1,PV2,P1),
    sas(S2,S3,PV2,PV3,P2).
sas(S1,S1,PV1,PV2,[]) :- unistr(PV1,PV2).
saval(S1,S2,PV,PV,[adv,P]) :- adv(S1,S2,P).
saval(S1,S2,PV1,PV2,P) :- saval1(S1,S2,PV1,[],PV2,P).
saval1(S1,S2,[tovo!PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[ntovo!PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[nstgo!PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[pn!PN]!PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[objectbe!PV],PV1,PV2,P) :- !,saval1(S1,S2,PV,PV1,PV2,P).
saval1(S1,S2,[OTHER!PV],PV1,PV2,P) :-
    !,saval1(S1,S2,PV,[OTHER!PV1],PV2,P).
saval1(S1,S2,[],[],P) :- !,fail.
saval1(S1,S2,[],PV,PV,[in-order-to,P]) :- tovo(S1,S2,nogap,pro,P).
saval1(S1,S2,[],PV,PV,P) :- pn(S1,S2,nogap,PP,P).
objectbe-verb(Sts) :- mem(objectbe,Sts).
nstgo-verb(Sts) :- mem(nstgo,Sts).
tovo-verb(Sts) :- mem(tovo,Sts).
nullobj-verb(Sts) :- mem(nullobj,Sts).
pn-verb(Sts,PP) :- mem([pn!PP],Sts).
vb(X,Y,Sts,Num,V) :- verb(X,Y,Sts,Num,V),!.
vb(X,Y,Sts,Num,V) :- ven(X,Y,Sts,V),!.
vb(X,Y,Sts,Num,V) :- ving(X,Y,Sts,V),!.
auxverb(V1,V2,Sts,Num,P) :- def(verb,P,aux,Sts),!,vrb(V1,V2,Num,P).
auxverb(V1,V2,Sts,Num,P) :- vrb(V1,V2,Num,P),def(verb,P,aux,Ost),@(X,Sts,Ost).
verb(V1,V2,Sts,Num,P) :- def(verb,P,Aux,Sts),!,vrb(V1,V2,Num,P).
verb(V1,V2,Sts,Num,P) :- vrb(V1,V2,Num,P),def(verb,P,Aux,Ost),@(X,Sts,Ost).
ven(V1,V2,Sts,P) :- vrb(V1,V2,en,P),def(verb,P,Aux,Sts),!.
ven(V1,V2,Sts,P) :- vrb(V1,V2,en,P),def(verb,P,Aux,Ost),@(X,Sts,Ost).
ving(V1,V2,Sts,P) :- vrb(V1,V2,ing,P),def(verb,P,Aux,Sts),!.
ving(V1,V2,Sts,P) :- vrb(V1,V2,ing,P),def(verb,P,Aux,Ost),@(X,Sts,Ost).
unistr(S,S) :- !.
unistr([E!S1],[E!S2]) :- unistr(S1,S2).
unistr([E!S1],S2) :- @(X,[E!Y],S2),@(X,Y,S3),!,unistr(S1,S3).

```

Appendix C: A lexicon for a fragment of English

vrbl([arrived!V],V,sg,arrive).
vrbl([arrived!V],V,pl,arriv ).
vrbl([arrive!V],V,pl,arrive).
vrbl([arrive!V],V,inf,arrive).
vrbl([seems!V],V,sg,seem).
vrbl([want!V],V,pl,want).
vrbl([want!V],V,inf,want).
vrbl([wants!V],V,sg,want).
vrbl([eat!V],V,inf,eat).
vrbl([eat!V],V,pl,eat).
vrbl([ate!V],V,sg,eat).
vrbl([ate!V],V,pl,eat).
vrbl([eats!V],V,sg,eat).
vrbl([looks!V],V,sg,look).
vrbl([look!V],V,pl,look).
vrbl([look!V],V,inf,look).
vrbl([like!V],V,inf,like).
vrbl([like!V],V,pl,like).
vrbl([likes!V],V,sg,like).
vrbl([be!V],V,inf,be).
vrbl([is!V],V,sg,be).
vrbl([are!V],V,pl,be).
vrbl([were!V],V,pl,be).
vrbl([does!V],V,sg,do).
vrbl([do!V],V,pl,do).
vrbl([did!V],V,sg,do).
vrbl([did!V],V,pl,do).
vrbl([fly!V],V,pl,fly).
vrbl([fly!V],V,inf,fly).
vrbl([flies!V],V,sg,fly).
vrbl([time!V],V,pl,time).
vrbl([time!V],V,inf,time).
def(verb,arrive,naux,[nullobj]).
def(verb,seem,naux,[tovo]).
def(verb,want,naux,[tovo,nstgo]).
def(verb,eat,naux,[nstgo]).
def(verb,like,naux,[nstgo]).
def(verb,be,aux,[objectbe]).
def(verb,do,aux,[nstgo]).
def(verb,look,naux,[nullobj,[pn,at]]).
def(verb,fly,naux,[nullobj,nstgo]).
def(verb,time,naux,[nstgo]).
noun([john!N],N,sg,john).
noun([mary!N],N,sg,mary).
noun([man!N],N,sg,man).
noun([men!N],N,pl,man).
noun([fish!N],N,X,fish).
noun([dog!N],N,sg,dog).
noun([dogs!N],N,pl,dog).
noun([monday!N],N,sg,monday).
noun([time!N],N,sg,time).
noun([flies!N],N,pl,fly).
noun([arrow!N],N,sg,arrow).
noun([planes!N],N,pl,plane).
adj([tall!A],A,tall).

adj([time|A],A,time).
 adj([happy|A],A,happy).
 adj([dangerous|A],A,dangerous).
 det([a|D],D,sg,a).
 det([an|D],D,sg,an).
 det([some|D],D,X,some).
 det([every|D],D,sg,every).
 det([all|D],D,pl,every).
 count([six|Q],Q,pl,six).
 to([to|T],T).
 pro([he|P],P,sg,he).
 pro([him|P],P,sg,he).
 pro([you|P],P,pl,you).
 whword([who|W],W,[tpos,who]).
 whword([which|W],W,[tpos,which]).
 whword([what|W],W,[tpos,what]).
 where([where|W],W,where).
 whques([where|W],W,where).
 whques([how|W],W,how).
 vrb([eaten|V],V,en,eat).
 vrb([arrived|V],V,en,arrive).
 vrb([liked|V],V,en,like).
 vrb([been|V],V,en,be).
 vrb([done|V],V,en,do).
 vrb([eating|V],V,ing,eat).
 vrb([being|V],V,ing,be).
 vrb([doing|V],V,ing,do).
 vrb([arriving|V],V,ing,arrive).
 vrb([flying|V],V,ing,fly).
 by([by|B],B,by).
 that([that|T],T).
 wh-that([that|T],T).
 wh-that([who|T],T).
 prep([to|P],P,to).
 prep([like|P],P,like).
 prep([on|P],P,on).
 prep([in|P],P,in).
 prep([at|P],P,at).
 adv([quickly|A],A,quickly).
 adv([early|A],A,early).
 adv([yesterday|A],A,yesterday).
 adv([dangerously|A],A,dangerously).

NYU COMPSCI TR-465
Strzalkowski, Tomek
Automated inversion of a
unification parser into a
unification... c.2

FOURTEEN DAYS

CAYLORD 142			PRINTED IN U S A

